**SignChat API: Sistema Backend para la transformación de mensajes de español a lenguaje de señas colombiano (LSC) como servicio para la aplicación SignChat**

**Harold Samuel Barrera Niño**

**William David Florez Londoño**

**Universidad de San Buenaventura, Sede Bogotá.**

**Facultad de Ingeniería.**

**Ingeniería de Sistemas**

**Bogotá, Colombia**

**2023**

**SignChat API: Sistema Backend para la transformación de mensajes de español a lenguaje de señas colombiano (LSC) como servicio para la aplicación SignChat**

**Harold Samuel Barrera Niño**

**William David Florez Londoño**

**Yamil Buenaño Palacios**

**Universidad de San Buenaventura, Sede Bogotá.**

**Facultad de Ingeniería.**

**Ingeniería de Sistemas**

**Bogotá, Colombia**

**2023**

Contenido

[INTRODUCCIÓN 2](#_Toc149056146)

[Capítulo 1 1](#_Toc149056147)

[1.1 Antecedentes 1](#_Toc149056148)

[1.2 Planteamiento del problema, Justificación y pregunta de Investigación 2](#_Toc149056149)

[1.3 Objetivo General 3](#_Toc149056150)

[1.4 Objetivos Específicos 3](#_Toc149056151)

[1.5 Alcances y Limitaciones 4](#_Toc149056152)

[1.6 Matriz de Marco Lógico 5](#_Toc149056153)

[1.6.1 Planificación 5](#_Toc149056154)

[1.6.2 Análisis 5](#_Toc149056155)

[1.6.3 Diseño 5](#_Toc149056156)

[1.6.4 Desarrollo 5](#_Toc149056157)

[1.6.5 Prueba 6](#_Toc149056158)

[1.6.6 Despliegue 6](#_Toc149056159)

[1.7 Marco Conceptual 7](#_Toc149056160)

[1.7.1 HTTP 7](#_Toc149056161)

[1.7.2 API 7](#_Toc149056162)

[1.7.3 API REST 8](#_Toc149056163)

[1.7.4 Web service 9](#_Toc149056164)

[1.7.5 Microservicios 9](#_Toc149056165)

[1.7.6 Integración microservicios 10](#_Toc149056166)

[1.7.7 Java 10](#_Toc149056167)

[1.7.8 PHP 11](#_Toc149056168)

[1.7.9 Spring 11](#_Toc149056169)

[1.7.10 Spring Boot 11](#_Toc149056170)

[1.8 Metodología 13](#_Toc149056171)

[1.8.1 Planificación 13](#_Toc149056172)

[1.8.2 Análisis 14](#_Toc149056173)

[1.8.3 Diseño 14](#_Toc149056174)

[1.8.4 Desarrollo 14](#_Toc149056175)

[1.8.5 Prueba 15](#_Toc149056176)

[1.8.6 Despliegue 15](#_Toc149056177)

[1.8.7 Metodología SCRUM 16](#_Toc149056178)

[1.9 Cronograma 17](#_Toc149056179)

[Capítulo 2. Desarrollo de Ingeniería 18](#_Toc149056180)

[2.1 Planificación 18](#_Toc149056181)

[2.1.1 Identificación de herramientas necesarias para la construcción de la aplicación. 18](#_Toc149056182)

[2.1.2 Estudio del comportamiento del sistema de mensajería en tiempo real de la aplicación SignChat. 20](#_Toc149056183)

[2.2 Análisis 22](#_Toc149056184)

[2.2.1 Análisis y creación de requerimientos funcionales. 22](#_Toc149056185)

[2.2.2 Estudiar la conexión a la base de datos. 23](#_Toc149056186)

[2.3 Diseño 24](#_Toc149056187)

[2.3.1 Instalación de tecnologías necesarias. 24](#_Toc149056188)

[2.3.1.1 IntelliJ 24](#_Toc149056189)

[2.3.1.2 Git 24](#_Toc149056190)

[2.3.1.3 Java 17 24](#_Toc149056191)

[2.3.1.4 Spring Boot 3.1.0 24](#_Toc149056192)

[2.3.1.5 Swagger UI 25](#_Toc149056193)

[2.3.1.6 Spring Data JPA 25](#_Toc149056194)

[2.3.1.7 H2 Database 25](#_Toc149056195)

[2.3.1.8 Lombok 25](#_Toc149056196)

[2.3.1.9 Spring Boot Devtools 25](#_Toc149056197)

[2.3.1.10 Websocket 25](#_Toc149056198)

[2.3.2 Desarrollo de la arquitectura de la base de datos a utilizar. 25](#_Toc149056199)

[2.3.3 Creación de diagrama de clases UML de la aplicación. 27](#_Toc149056200)

[2.3.4 Desarrollo de la arquitectura de integración. 28](#_Toc149056201)

[2.4 Desarrollo 30](#_Toc149056202)

[2.4.1 Creación de la base de datos. 30](#_Toc149056203)

[2.4.2 Implementación de las clases establecidas en los diagramas. 34](#_Toc149056204)

[2.4.2.1 Servicios e Interfaces. 34](#_Toc149056205)

[2.4.2.2 Configuraciones. 41](#_Toc149056206)

[2.4.3 Codificación de las funcionalidades requeridas. 43](#_Toc149056207)

[2.4.3.1 Controlador de lista de contactos. 43](#_Toc149056208)

[2.4.3.2 Controlador de usuarios. 44](#_Toc149056209)

[2.4.3.3 Controlador de mensajes. 45](#_Toc149056210)

[2.4.3.4 Controlador del teclado LSC. 50](#_Toc149056211)

[2.4.3.5 Controlador del websocket. 56](#_Toc149056212)

[2.4.4 Creación de páginas web de prueba. 57](#_Toc149056213)

[2.4.4.1 Php 57](#_Toc149056214)

[2.4.4.2 JavaScript 58](#_Toc149056215)

[2.4.5 Retroalimentación y corrección de errores. 65](#_Toc149056216)

[2.4.5.1 Problema en la configuración de CORS 65](#_Toc149056217)

[2.4.5.2 Problema en la conexión con los websockets 66](#_Toc149056218)

[2.4.5.3 Problema en la implementación utilizando JavaScript 66](#_Toc149056219)

[2.4.5.4 Problema en la implementación utilizando PHP 70](#_Toc149056220)

[2.5 Pruebas 71](#_Toc149056221)

[2.5.1 Comprobación de tecnologías necesarias. 71](#_Toc149056222)

[2.5.2 Comprobación del comportamiento de la base de datos. 73](#_Toc149056223)

[2.5.3 Comprobación de funcionalidad. 75](#_Toc149056224)

[2.6 Despliegue 76](#_Toc149056225)

[2.6.1 Creación del plugin. 76](#_Toc149056226)

[2.6.2 Lanzamiento del microservicio. 77](#_Toc149056227)

[Capítulo 3. Análisis de Resultados. 81](#_Toc149056228)

[3.1 Microservicio de SignChat. 81](#_Toc149056229)

[3.2 Página Web de prueba. 86](#_Toc149056230)

[CONCLUSIONES 91](#_Toc149056231)

[REFERENCIAS 93](#_Toc149056232)

[Anexo 1. Instalación de IntelliJ 95](#_Toc149056233)

[Anexo 2. Instalación de GIT 95](#_Toc149056234)

[Anexo 5. Descarga de Swagger UI 102](#_Toc149056235)

[Anexo 6. Implementación Spring Data JPA 102](#_Toc149056236)

[Anexo 7. Implementación H2 Database 103](#_Toc149056237)

[Anexo 8. Lombok 103](#_Toc149056238)

[Anexo 9. Implementación Spring Boot Devtools 104](#_Toc149056239)

[Anexo 10. Implementación Websocket 104](#_Toc149056240)

[Anexo 11. Manual 105](#_Toc149056241)

**Lista de tablas**

[Tabla 1. Requerimientos funcionales. 23](#_Toc149058272)

[Tabla 2. Resultado del producto. 76](#_Toc149058273)

**Lista de Figuras**

Ilustración 1. Diagrama SCRUM del proyecto. 16

Ilustración 2. Cronograma. 17

Ilustración 3. Comportamiento del sistema de mensajería en tiempo real de SignChat. 21

Ilustración 4. Websocket. 22

Ilustración 5. Base de datos H2. 24

Ilustración 6. Arquitectura de la base de datos. 26

Ilustración 7.Diagrama UML de la aplicación. 27

Ilustración 8. Arquitectura de Integración. 29

Ilustración 9. Teclado de señas virtual 58

Ilustración 10. Teclado de señas virtual 59

Ilustración 11. Mensaje de conexión exitoso 61

Ilustración 12. Pagina HTML que implementa teclado mediante JavaScript 62

Ilustración 13. Pagina HTML #2 que implementa teclado usando JavaScript 63

Ilustración 14. Conexión a endpoint usando SockJS 72

Ilustración 15. Conexión websocket implementando StompJS 72

Ilustración 16. Swagger funcionalidades 1 73

Ilustración 17. Swagger funcionalidades 2 73

Ilustración 18. Configuración proyecto dependencia H2 74

Ilustración 19. Comprobación conexión de base de datos 74

Ilustración 20. Funcionamiento de Base de datos 75

Ilustración 21. Visualización de teclado virtual. 76

Ilustración 22. Plugin de SignChat. 77

Ilustración 23. Render. 77

Ilustración 24. Configuraciones hechas en Render. 79

Ilustración 25. Microservicio lanzado en Render. 79

Ilustración 26. Microservicio con Host en Render. 80

Ilustración 27. Pantalla principal. 81

Ilustración 28. Swagger UI. 82

Ilustración 29. Swagger UI, funcionalidades. 82

Ilustración 30. Swagger UI, Esquemas. 83

Ilustración 31. Swagger UI, prueba de API REST. 83

Ilustración 32. Inicio de sesión de base de datos. 84

Ilustración 33. Conexión exitosa de base de datos. 84

Ilustración 34. Interfaz de Base de datos. 84

Ilustración 35. Websocket. 85

Ilustración 36. Teclado virtual LSC. 85

Ilustración 37. Teclado LSC en acción. 86

Ilustración 38. Mensaje de conexión con base de datos temporal 86

Ilustración 39. Teclado de señas virtual, con espacio de texto vacío 87

Ilustración 40. Teclado de señas virtual, con espacio de texto lleno 88

Ilustración 41. Pagina HTML de prueba que implementa teclado y su funcionamiento 89

Ilustración 42. Pagina HTML implementa más input 90

Ilustración 43. Instalador Java para Windows. 96

Ilustración 44. Instalador Java para Mac. 97

Ilustración 45. Spring Initialzr 98

Ilustración 46. Configuración del proyecto. 99

Ilustración 47. Funcionamiento Swagger 105

Ilustración 48. Funcionamiento Swagger 105

Ilustración 49.Swagger estructura de Base de Datos 107

**Nomenclatura**

**API:** Mecanismo para la comunicación entre dos componentes al utilizar definiciones y protocolos compartidos.

**API REST:** Arquitectura de software usada para la creación de páginas web.

**Spring Boot**: Framework de desarrollo de aplicaciones en lenguaje Java.

**Java**: Lenguaje de programación orientado a objetos usado para el desarrollo del proyecto.

**PHP**: Lenguaje de desarrollo web principalmente usado para la comunicación con base de datos.

**Web Service:** Protocolo de comunicación para intercambio de datos entre dispositivos electrónicos vía red.

**Lombok:** Biblioteca en lenguaje Java para ahorrar código y así ser más ligera y eficiente.

**Microservicio**: Conjunto de servicios ejecutados de forma conjunta mediante un API.

**H2**: Sistema administrador de bases de datos relacionales programado en Java.

**Swagger UI**: Herramienta que nos permite ver y probar de manera grafica desde Spring Boot.

**SockJS**: Biblioteca JavaScript para la comunicación bidireccional con página web y servidor. implementación de websocket.

**StompJS**: Utilizado para la conexión y envió de mensajes al servidor websocket.

# INTRODUCCIÓN

La razón por la cual se han planeado el desarrollo de este proyecto es continuar con el proyecto de la aplicación SignChat. Las funcionalidades que ofrece el aplicativo para ser convertido en un microservicio para ser utilizado por otras aplicaciones que permita ampliar el área donde esta pueda ser utilizada, como lo son las diferentes aplicaciones de chat de texto, siendo este funcional en la comunicación de aplicaciones externas. La metodología elegida es el desarrollo en cascada, donde se podrán realizar varias actividades en conjunto. Esta metodología se ha elegido debido a que ofrece una línea de tiempo mejor organizada y definida en los tiempos de entrega, además, de la capacidad de modificar los horarios. El resultado final esperado es: la creación de o modificación de la aplicación SignChat para prestar la funcionalidad como microservicio.

# Capítulo 1

## 1.1 Antecedentes

Los trabajos investigados y expuestos a continuación hacen referencia a diferentes proyectos que tienen relación con la propuesta de nuestro trabajo de investigación en diferentes maneras:

* Certificado ATICA 2020, “SignChat: Sistema de chat para personas con discapacidad visual parcial y en condición de sordera” por los Jorge Saad, Oscar Erazo y Sebastián Sánchez y el profesor Yamil Buenaño Palacios y Andrés Sánchez
* Tesis “Refactorización de la aplicación SignChat: Incorporación de mecanismos de accesibilidad para personas con discapacidad visual parcial y mejora de la usabilidad para las personas en condición de sordera” por Erazo Moncayo Oscar Hernán, Saad Ramírez Jorge Felipe y Sánchez Cid Sebastián, los directores de proyecto Ing. Andrés Armando Sánchez Martin, Ing. Yamil Buenaño Palacios

El proyecto toma como base la aplicación SignChat, un sistema de mensajería en tiempo real que utiliza Firebase como servicio de Backend. Con Firebase, logra él envió de mensajes en tiempo real y también el hosting de la base de datos en la nube.

## 1.2 Planteamiento del problema, Justificación y pregunta de Investigación

En la actualidad un porcentaje de la población actualmente sufre de ciertas condiciones que hacen que la persona tenga problemas de audición y no pueden interactuar de la mejor manera con las demás personas, razón por la cual se creó la aplicación SignChat la cual es una aplicación para la comunicación que utiliza lenguaje de señas colombiano para comunicarse con otra persona, esta aplicación causo un impacto por la novedad que proponía en Colombia pues este era un proyecto que cubría la necesidad de facilitar la comunicación con las personas que sufren de discapacidad de audición.

Pese a que el chat de la aplicación es una característica destacable no es utilizada por otras aplicaciones, pues la característica de la aplicación es el chat que permite una fácil comunicación entre usuarios, pero esto es algo propio de la aplicación por lo cual no es algo que esté disponible en otras aplicaciones, por lo cual, en este trabajo se busca la creación de un servicio utilizando como base la aplicación donde sea posible la comunicación en esta diferente aplicación al utilizar el servicio, por tanto ¿Cómo mejorar el rendimiento y escalabilidad de la aplicación SignChat con la implementación de Web Service?

## 1.3 Objetivo General

Desarrollar un API REST para la traducción de mensajes de español a lenguaje de señas colombiano (LSC) para la integración entre dos sistemas la aplicación SignChat.

## 1.4 Objetivos Específicos

* Diseñar un documento donde se exponga los pasos a seguir para el desarrollo de un microservicio
* Establecer un cronograma de trabajo utilizando metodología de cascada para organización de desarrollo de tareas
* Definir requerimientos funcionales, de rendimiento y seguridad del API REST de SignChat
* Evaluar tecnologías y arquitecturas más adecuada para la implementación del API REST que trabaje en tiempo real.
* Diseñar el modelo de datos y los contratos REST que cumplan con los requerimientos, arquitectura y tecnologías seleccionadas.
* Codificar el API REST diseñado con estándares de ingeniería de software y seguridad de la información.
* Implementar un protocolo de pruebas de carga y estrés sobre el API REST desarrollado.

## 1.5 Alcances y Limitaciones

Desarrollaremos una aplicativo cuya función será ser un microservicio que prestará la función de comunicación, conversión de mensajes que se encuentra en la aplicación de SignChat, este será probado en una aplicación de prueba para él envió de mensajes (de SO Android) y la comunicación esto para tener un entorno de pruebas controlado entregando documentación que reporte el desarrollo del trabajo.

## 1.6 Matriz de Marco Lógico

Usaremos el ciclo de vida de desarrollo de software para la realización del proyecto. Esto quiere decir que el consta de 6 fases: planificación, análisis, diseño, desarrollo, prueba y despliegue. A continuación, la serie de pasos que se seguirá en cada fase:

### 1.6.1 Planificación

* Identificación de herramientas necesarias para la construcción de la aplicación.
* Estudio del comportamiento del sistema de mensajería en tiempo real de la aplicación SignChat.

### 1.6.2 Análisis

* Análisis y creación de requerimientos funcionales.
* Estudiar la conexión a la base de datos.

### 1.6.3 Diseño

* Instalación de tecnologías necesarias.
* Desarrollo de la arquitectura de base de datos a utilizar.
* Creación de diagrama de clases UML de la aplicación.

### 1.6.4 Desarrollo

* Creación de la base de datos.
* Implementación de las clases establecidas en los diagramas.
* Codificación de las funcionalidades requeridas.
* Creación de paginas web de prueba.
* Retroalimentación y corrección de errores.

### 1.6.5 Prueba

* Comprobación de tecnologías necesarias.
* Comprobación del comportamiento de la base de datos.
* Comprobación de funcionalidad.

### 1.6.6 Despliegue

* Despliegue de la base de datos
* Lanzamiento del microservicio.

## 1.7 Marco Conceptual

### 1.7.1 HTTP

El http (del inglés HyperText Transfer Protocol o Protocolo de Transferencia de Hiper Textos) es el protocolo de transmisión de información de la World Wide Web, es decir, el código que se establece para que el computador solicitante y el que contiene la información solicitada puedan “hablar” un mismo idioma a la hora de transmitir información por la red.

Este protocolo sirve para comunicar una página web con la computadora. El usuario realmente no puede ver dicha comunicación, pero lo que ocurre detrás del protocolo es una serie de peticiones que son pedidas por el usuario, esto puede ser por ejemplo el botón de "iniciar sesión" de alguna página como Netflix. Por lo general, funciona transmitiendo los datos del usuario, que varían de la aplicación, pero siempre se encuentra el dominio origen (el usuario) y dominio destino (el servicio web).

Este protocolo cuenta con varios métodos, los cuales son:

* **GET:** utilizado para el acceso a recursos o información que no necesita ser modificada
* **POST:** utilizado para él envió de información al servidor como puede ser información de un formulario
* **PUT:** utilizado para reemplazar información o características existentes en el sistema
* **DELETE:** utilizado para borrar o eliminar información específica de la base de datos o servidor

### 1.7.2 API

Un API (Interfaz de programación de aplicaciones) es un mecanismo para la comunicación entre dos componentes al utilizar definiciones y protocolos compartidos. Por lo general estas funcionan al usar una estructura cliente servidor para ser realizar peticiones de ejecuciones de funcionalidades, existen diferentes tipos de API como API SOAP, API de RPC, API WebSocket y la más utilizada API REST

### 1.7.3 API REST

REST es una arquitectura de software usada para la creación de páginas web. Es la arquitectura más usada hoy en dia debido a su eficiencia y rapidez con la transmisión de los datos. Esta arquitectura envía los datos mediante HTTP, los cuales pueden llegar en varios formatos como puede ser HTML, PHP, Python o incluso texto sin formato. Sin embargo, el formato para los datos que más se usa junto a REST es JSON. JSON (JavaScript Object Notation) es un lenguaje de programación popular para describir los datos, puesto que este, puede ser comprendido tanto por humanos como por maquinas.

Una API son un conjunto de protocolos diseñados por la propia empresa para que sus usuarios puedan usar su servicio o aplicación.

Una API REST es una interfaz de programación que permite la interacción con los servicios web. Este se encuentra limitado por la arquitectura REST. Dicha arquitectura tiene lo siguiente:

* Arquitectura cliente-servidor compuesta de clientes, servidores y recursos, con la gestión de solicitudes a través de HTTP.
* Comunicación entre el cliente y el servidor sin estado, lo cual implica que no se almacena la información del cliente entre las solicitudes de GET y que cada una de ellas es independiente y está desconectada del resto.
* Datos que pueden almacenarse en caché y optimizan las interacciones entre el cliente y el servidor.

### 1.7.4 Web service

Los Web Service son métodos de comunicación entre dispositivos electrónicos vía red, esto son colecciones de protocolos y estándares para intercambiar datos entre aplicaciones o sistemas independientemente del lenguaje que se utilize debido al uso de los estándares abiertos, unos ejemplos son que los sistemas de mensajes se utilizan XML estandarizados como el XML-RPC (Remote Procedure Call) que permite la ejecución de código de forma remota. Componentes de web services.

* Simple Object Access Protocol: formato de envió de mensajes y comunicación en el internet con el protocolo para el acceso al Web Service.
* Web Services Description Language: lenguaje basado en XML para el acceso a los Web Services además este es parte integral del UDDI.
* Universal Description, Discovery and Integration: directorio el cual es utilizado para la descripción, publicación y búsqueda de servicios web, este utiliza el WSDL para la comunicación con el SOAP.

### 1.7.5 Microservicios

Los microservicios son diversos elementos o procesos diseñados para realizar una serie de procesos específicos los cuales se desempeñan de forma independientes, pero son ejecutados en conjunto para realizar una tarea a diferencia de la arquitectura monolítica donde un solo elemento realiza todos los procedimientos, las características que posee es que esta brinda la oportunidad de facilitar el mantenimiento de los elementos que conforman esta tarea.

### 1.7.6 Integración microservicios

Como ya se explicó los microservicios son procesos que realizan actividades específicas de forma independiente de la aplicación, el proceso de integración en este proyecto se destinará en el desarrollo de una aplicación desarrollada en un lenguaje que permita ser utilizada por diferentes dispositivos móviles con un sistema operativo Android (este fue el elegido para el desarrollo de este proyecto), la integración de este servicio se realiza al directamente hacer uso del microservicio que envía una solicitud a la base de datos o a través de una API de conexión que realiza actividades específicas y posteriormente se conecta a un microservicio el cual realiza peticiones específicas a la base de datos, con este breve resumen del recorrido en un proceso que utiliza un microservicio se evidencia que los microservicios se ubican en un punto medio entre la aplicación y la base de datos siendo este exterior a la aplicación por lo que su implementación no afecta el funcionamiento básico de la aplicación base.

### 1.7.7 Java

Java es un lenguaje de programación orientado a objetos multiplataforma (software, aplicaciones, sistemas operativos, software empresarial, etc.), este lenguaje es uno de los más populares y utilizados a nivel mundial por el largo historial de aplicaciones desarrolladas con este lenguaje, por lo cual este será utilizado en el software de Android Studio para realizar procedimientos.

### 1.7.8 PHP

El lenguaje PHP está destinado al desarrollo back-end utilizado para realizar consultas ingresar información a la base de datos entre otras acciones de “web service”. Este un lenguaje utilizado para la comunicación entre las aplicaciones y la base de datos al recopilar y manipular la información para ser enviada a la base de datos

### 1.7.9 Spring

La tecnología Spring es un framework de código abierto destinada al desarrollo para la creación de software de gran complejidad pues está pensado para aplicaciones con las características de ser complejas, que estes distribuidas en diferentes servidores, que se desplieguen en redes corporativas o conectadas en la nube además estas se centran en el manejo de datos (las funcionalidades se destinan al manejo de los mismos manteniendo su integridad, confidencialidad y disponibilidad de los mismos), entre otras características, esto utilizando lenguaje de programación JAVA, este framework da soporte para Groovy(lenguaje de programación implementado en plataformas JAVA y su sintaxis es similar a Python o Ruby) y Kotlin(lenguaje de programación estatico creado por JetBrains).

### 1.7.10 Spring Boot

Spring Boot es un proyecto basado en tecnologías Spring la cual junta diferentes componentes y herramientas para la creación de servicios de forma rápida y eficiente. Al usar configuraciones preestablecidas y librerías de desarrollo, otras aplicaciones las cuales facilita el desarrollo son aquellas que no depende de la conexión a un servidor o recursos externo (aplicaciones autocontenidas), también utilizado en el desarrollo de servicios, microservicios o aplicaciones REST. Las características principales son su fácil uso, acceso a librerías de terceros, opción de conexión a cloud y la facilidad para realizar pruebas de funcionalidad.

## 1.8 Metodología

Se utilizó el ciclo de vida de desarrollo de software para el desarrollo del proyecto con características de la metodología SCRUM. Las características que utilizaremos son el uso del Sprint Backlog para generar los requisitos del proyecto, Sprint para la codificación de este y los Sprint Review para revisar los resultados después de cada fase.

Este proyecto consta de 6 fases: planificación, análisis, diseño, desarrollo, prueba y despliegue.

1. **Planificación:** Estudio del caso y herramientas utilizadas.
2. **Análisis:** Comportamiento y funcionamiento del plugin junto con SignChat.
3. **Diseño:** Creación de diagramas UML y arquitectura de Software.
4. **Desarrollo:** Codificación del plugin.
5. **Prueba:** Comprobación de requerimientos y comportamiento del software.
6. **Despliegue:** Lanzamiento del plugin.

### 1.8.1 Planificación

En esta fase declaramos las herramientas utilizadas en el plugin, además de explicar cómo implementaremos el sistema de mensajería en tiempo real de SignChat.

* **Tareas**
  + Identificación de herramientas necesarias para la construcción de la aplicación.
  + Estudio del comportamiento del sistema de mensajería en tiempo real de la aplicación SignChat.
* **Entregables**
  + Documento explicando el sistema de mensajería de SignChat.

### 1.8.2 Análisis

En esta fase establecemos los requerimientos funcionales que debe tener el plugin y el funcionamiento de la base de datos.

* **Tareas**
  + Análisis y creación de requerimientos funcionales.
  + Estudiar la conexión a la base de datos.
* **Entregables**
  + Documento de requerimientos.

### 1.8.3 Diseño

En esta fase mostramos las tecnologías que usamos en el plugin, los diagramas UML del software y de la base de datos.

* **Tareas**
  + Instalación de tecnologías necesarias.
  + Desarrollo de la arquitectura de la base de datos a utilizar.
  + Creación de diagrama de clases UML de la aplicación.
  + Desarrollo de la arquitectura de integración.
* **Entregables**
  + Documento con los diagramas creados.

### 1.8.4 Desarrollo

En esta fase se realiza toda la codificación del plugin.

* **Tareas**
  + Creación de la base de datos.
  + Implementación de las clases establecidas en los diagramas.
  + Codificación de las funcionalidades requeridas.
  + Creación de páginas web de prueba.
  + Retroalimentación y corrección de errores.
* **Entregables**
  + Repositorios de proyecto de Spring Boot y las páginas web de prueba.

### 1.8.5 Prueba

En esta fase se verifica que toda la codificación del plugin esté funcionando correctamente.

* **Tareas**
  + Comprobación de tecnologías necesarias.
  + Comprobación del comportamiento de la base de datos.
  + Comprobación de funcionalidad.
* **Entregables**
  + Esta fase no tiene entregables.

### 1.8.6 Despliegue

En esta fase se crea el archivo descargable del plugin.

* **Tareas**
  + Creación del plugin.
  + Lanzamiento del microservicio.
* **Entregables**
  + Archivo JAR del plugin.
  + URL publica para el uso del plugin.

## 1.8.7 Metodología SCRUM

La metodología SCRUM se utilizó para obtener el mejor resultado posible trabajando en equipo. Para este propósito se creó el Sprint Backlog o los requisitos funcionales del proyecto para tener en cuenta las tareas pendientes. Se desarrollo reuniones diarias con duraciones de 15 minutos aproximadamente, esto con el fin de retroalimentar y analizar los resultados buenos o malos de los Sprints anteriores. Por último, se realiza un Sprint Review con el profesor líder de semillero. Según el resultado de la revisión se decidirá si se continua a la siguiente fase, si se debe corregir o realizar cambios como de observa en la Ilustración 2.

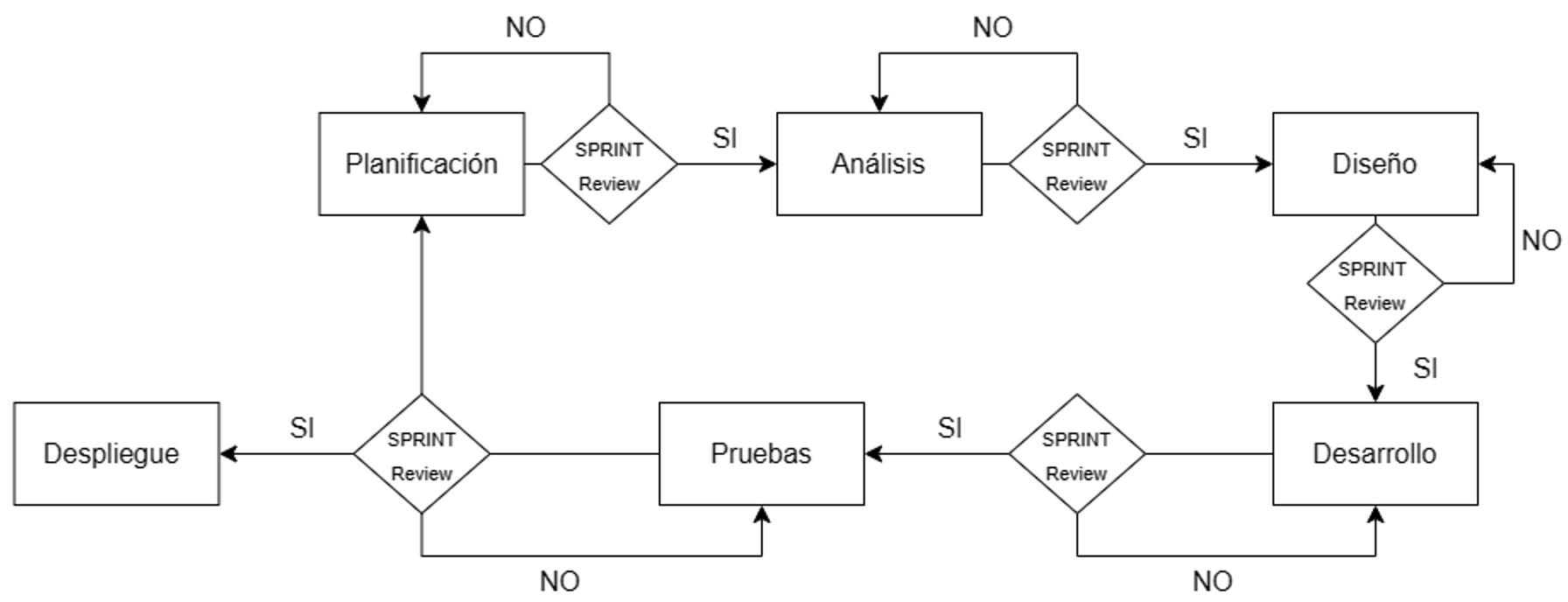


Ilustración 1. Diagrama SCRUM del proyecto.

Las fases del proyecto se harán de manera lineal para optimizar el tiempo de desarrollo del proyecto.

## 1.9 Cronograma

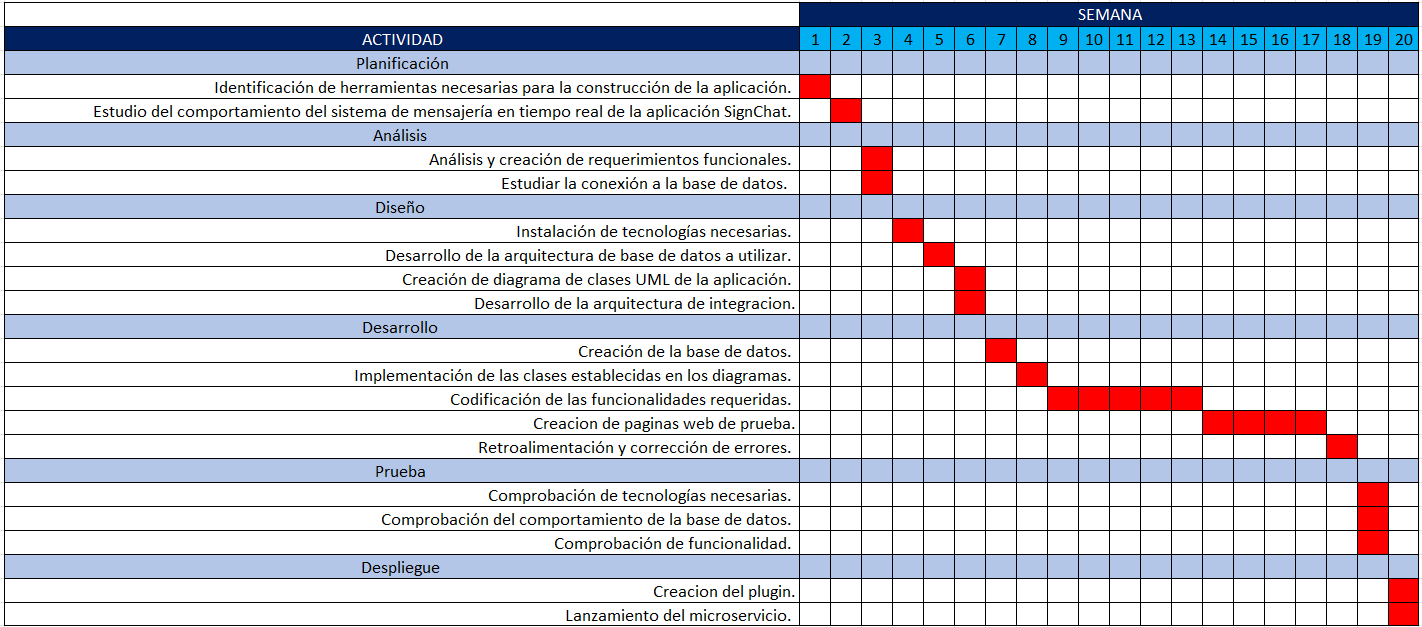


Ilustración 2. Cronograma.

# Capítulo 2. Desarrollo de Ingeniería

## 2.1 Planificación

### 2.1.1 Identificación de herramientas necesarias para la construcción de la aplicación.

Hay varias herramientas que vamos a utilizar en el transcurso del proyecto. Estas herramientas son el lenguaje de programación, el framework de trabajo, la base de datos, el editor de código y la aplicación con la que haremos las pruebas de la API REST.

El lenguaje de programación siempre es importante definirlo, ya que en este se construirá todo el proyecto. Para la creación del microservicio consideramos muchas opciones, pero se llegó a la conclusión que la mejor de estas opciones era usar el lenguaje de programación con el que fue construido SignChat, es decir, Java. Este lenguaje puede construir aplicaciones que sirven en cualquier sistema operativo gracias al JVM (Java Virtual Machine), una de las características que se buscan al desarrollar este microservicio.

Si vamos a utilizar Java, el mejor framework para lograr lo que buscamos es Spring Boot. Spring Boot es un framework de Java que nos sirve para construir aplicaciones Fullstack, pero, sobre todo, para la construcción de microservicios y ser conectados entre ellos para crear una gran aplicación con posibilidades de seguir escalando. También cuenta con una gran variedad de funciones, como controladores REST, seguridad, manejo y desarrollo de bases de datos con el modelo MVC (Model View Controller), entre otros. Estas funcionalidades pueden ser implementadas fácilmente puesto que Spring Boot usa repositorios Maven o Gradle (en nuestro caso, utilizaremos Maven).

La base de datos del microservicio es algo a destacar, puesto que, para nuestros objetivos, realmente no es necesario una base de datos. Sin embargo, se puede dar la ocasión que el usuario que quiera implementar nuestro servicio necesite de una base de datos, o tener de algún lugar donde guardar su información ya que posiblemente aún no ha creado su base de datos. Para esos casos donde el usuario quiere probar el servicio, implementaremos a nuestro proyecto Spring Boot la base de datos H2. Esta es una base de datos relacional, al igual que la base de datos de SignChat, construida en Java y para ser usadas en aplicaciones con el mismo lenguaje. H2 se usa, en lo general, cuando estas todavía desarrollando tu producto y por el momento no buscas que escale demasiado, lo cual nos conviene debido a que la base de datos dentro de nuestro microservicio apenas es utilizada, y si ya el usuario que está implementando nuestro servicio quiere una base de datos que escale mejor, debe ser el quien decida qué base de datos implementar en su aplicación, puesto que al fin y al cabo, el microservicio solo otorgara la traducción de mensajes de LSC a español y viceversa en tiempo real.

Hay muchas herramientas que nos pueden ayudar al momento de editar código. Tenemos los editores de código más famosos como Visual Studio Code o Sublime Text, pero cuando codificamos en Java, un editor de código común y corriente no suele ser suficiente. En esos casos es cuando utilizamos IDEs (Entornos de desarrollo Integrados), los cuales son editores de texto, pero con herramientas y funcionalidades extras ya implementadas para facilitar el desarrollo. Java tiene muchos IDEs para elegir, tenemos NetBeans, Eclipse o incluso hay un IDE propio de Spring Boot, pero la decisión final fue utilizar el IDE más conocido actualmente por su gran eficiencia y ese es IntelliJ Idea. Este IDE es desarrollado por Jetbrains, una empresa conocida por desarrollar varios IDEs para varios lenguajes, como JavaScript, Ruby, Php, entre otros.

Para finalizar, debemos tener alguna forma de comprobar que nuestro microservicio y sus funcionalidades trabajan correctamente. Para probar una API REST, por lo general siempre se necesita una página web para hacer la prueba, y aunque si se construirá algunas páginas web de prueba, el construir una página web simple lleva tiempo por lo que una solucion a ese problema es utilizar herramientas REST. Una herramienta REST nos ahorra la parte de tener que construir una página web para hacer la prueba directamente en las funcionalidades del microservicio. Existen varias herramientas y todas cumplen con el requisito, pero en nuestro caso utilizaremos Postman debido a que es la herramienta con la que mejor estamos familiarizados. También implementaremos en nuestro microservicio Swagger UI, que es una interfaz que nos permite ver y probar de manera grafica desde el propio Spring Boot las rutas de nuestra API REST.

### 2.1.2 Estudio del comportamiento del sistema de mensajería en tiempo real de la aplicación SignChat.

La aplicación SignChat utiliza Firebase Realtime Database para el manejo de su sistema de mensajería en tiempo real. Esto es una funcionalidad del BaaS Firebase, el cual es una base de datos NoSQL que guarda los datos en formato JSON, para que los clientes que estén usando la aplicación puedan ver los datos sincronizados en tiempo real.

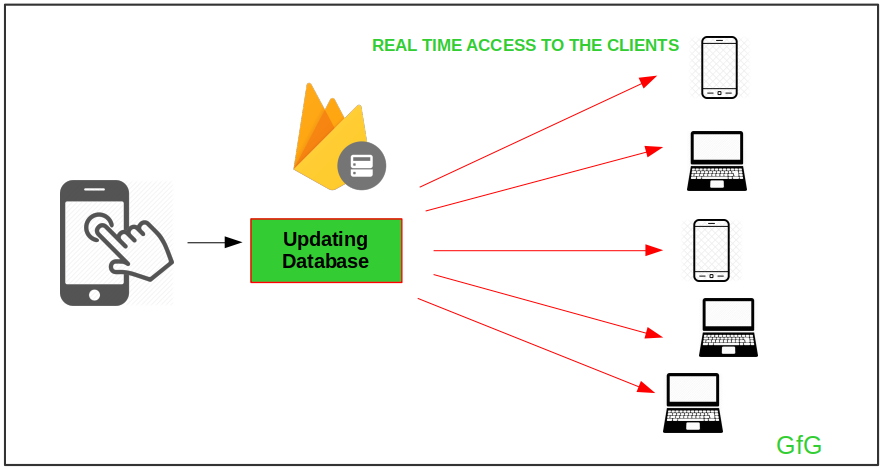


Ilustración 3. Comportamiento del sistema de mensajería en tiempo real de SignChat.

Nuestro microservicio no puede contar con un BaaS como Firebase, puesto que no es una aplicación completa, es un servicio que se puede utilizar en cualquier aplicación. Para solucionar este problema, utilizaremos websockets. Un websocket es una tecnología que permite abrir una sesión de conexión con el cliente y el servidor, esta sesión abre un flujo de información la cual, a diferencia de una conexión HTTP, la conexión no se cerrara después de una petición, seguirá activa hasta que el usuario cierre la sesión. De esta forma, podemos implementar el manejo y muestra de información en tiempo real.

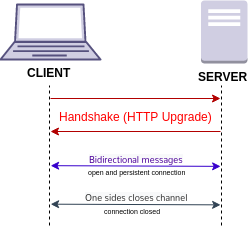


Ilustración 4. Websocket.

En Spring Boot, podemos implementar websockets con ayuda de la tecnología SockJS y StompJS. SockJS es la tecnología que nos ayudara con la implementación de websockets y StompJS es la forma para enviar mensajes al servidor websocket.

## 2.2 Análisis

### 2.2.1 Análisis y creación de requerimientos funcionales.

Para los requerimientos funcionales, definimos la siguiente tabla.

|  |  |  |
| --- | --- | --- |
| Código | Titulo | Descripción |
| RF01 | Crear y enviar mensaje. | El microservicio debe poder crear un mensaje y enviarlo de manera inmediata al receptor de dicho mensaje. |
| RF02 | Traducción de mensajes de imagen a texto. | El microservicio traducirá los mensajes escritos en el Lenguaje de Señas Colombiano a español latino, y a su vez, poder ser guardados en la base de datos en un formato valido desde el lado del servidor. |
| RF03 | Traducción de mensajes de texto a imagen. | El microservicio traducirá los mensajes escritos en español al Lenguaje de Señas Colombiano, desde el lado del cliente. |
| RF04 | Mostrar mensajes en tiempo real. | El microservicio contara con websockets, para poder realizar la transmisión de mensajes en tiempo real. |
| RF05 | Construcción de teclado de Lenguaje de Señas Colombiano. | El microservicio construirá el teclado de Lenguaje de Señas Colombiano que se encuentra en la aplicación SignChat en código HTML para poder ser llamado y usado en cualquier página web al antojo del desarrollador. |
| RF06 | Implementación de base de datos para pruebas. | El microservicio tendrá instalado la base de datos H2 para realizar pruebas de funcionalidades. |

Tabla 1. Requerimientos funcionales.

### 2.2.2 Estudiar la conexión a la base de datos.

El microservicio no requiere una base de datos para cumplir con sus requerimientos funcionales. En cualquier microservicio, quien desee usarlo para aplicarlo en una empresa, deberá implementar una base de datos para almacenar la información de los mensajes, los usuarios, sus contactos, fecha y hora de registro de mensajes, entre otros detalles que se deben guardar en una aplicación de mensajería en tiempo real. Esto por dar un ejemplo, porque este microservicio se puede usar en otras aplicaciones aparte de sistemas de mensajería en tiempo real, puesto que está construido para que pueda ser usado en cualquier página web.

A pesar de todo, algunos desarrolladores les gusta probar las tecnologías antes de usarlas en producción. Debido a ese motivo, implementamos en el microservicio SignChat la base de datos H2. H2 es un sistema de base de datos embebida en el microservicio, es decir, que está en la aplicación y no requiere mantenimiento para su funcionamiento. H2 siempre ha sido usado para la realización de pruebas de software, debido a que su escalabilidad no es muy buena. Sin embargo, este problema realmente no afecta nuestro objetivo final. Si el cliente que está usando nuestro microservicio requiere de una base de datos con escalabilidad, debe ser el quien lo implemente, con sus relaciones, tablas y entidades.

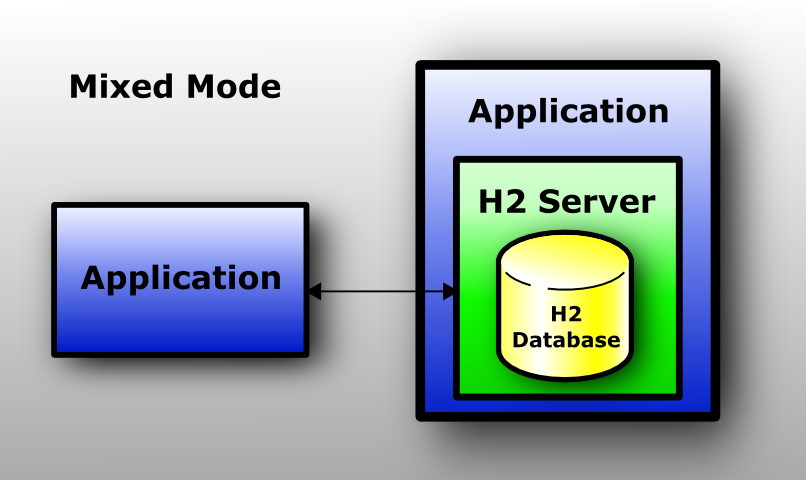


Ilustración 5. Base de datos H2.

## 2.3 Diseño

### 2.3.1 Instalación de tecnologías necesarias.

#### 2.3.1.1 IntelliJ

Información sobre la instalación del programa IntelliJ IDE Comunity en el Anexo 1.

#### 2.3.1.2 Git

Información sobre la instalación del programa GIT en el Anexo 2.

#### 2.3.1.3 Java 17

Información y procedimientos sobre la instalación del JAVA 17 en el Anexo 3.

#### 2.3.1.4 Spring Boot 3.1.0

Información sobre la instalación y configuración de Spring Boot en el Anexo 4.

#### 2.3.1.5 Swagger UI

Información sobre la instalación e implementación de la dependencia Swagger en el anexo 5.

#### 2.3.1.6 Spring Data JPA

Información sobre la instalación e implementación de la dependencia Spring Data JPA en el anexo 6.

#### 2.3.1.7 H2 Database

Información sobre la instalación e implementación de la dependencia H2 Database en el anexo 7.

#### 2.3.1.8 Lombok

Información sobre la instalación e implementación de la dependencia Lombok en el anexo 8.

#### 2.3.1.9 Spring Boot Devtools

Información sobre la instalación e implementación de la dependencia Spring Boot Devtools en el anexo 9.

#### 2.3.1.10 Websocket

Información sobre la instalación e implementación de la dependencia Spring Boot Devtools en el anexo 10.

### 2.3.2 Desarrollo de la arquitectura de la base de datos a utilizar.

El microservicio, aunque no requiere de base de datos, si es importante al menos dejar una arquitectura definida para evitar futuros errores. Sin embargo, solo necesitamos realmente un modelo, que es el modelo para los mensajes. Por ese motivo, decidimos construir dos modelos más en la base de datos: el modelo usuario y lista de contactos. Estos dos modelos, aunque nunca usados para el funcionamiento del teclado de SignChat, servirá si a futuro, el cliente que use el microservicio necesita un lugar donde guardar sus usuarios con su lista de contacto o al menos para tener una plantilla de cómo se debería ver y comportar sus usuarios en su aplicación si utilizara nuestro microservicio, aunque realmente la plantilla será una recomendación más que una regla. Además, recordar que esta base de datos es solo para pruebas únicamente, ya que la escalabilidad de la base de datos de H2 no es muy buena.

Con todas esas especificaciones, nuestra arquitectura de base de datos se vera de la siguiente forma:

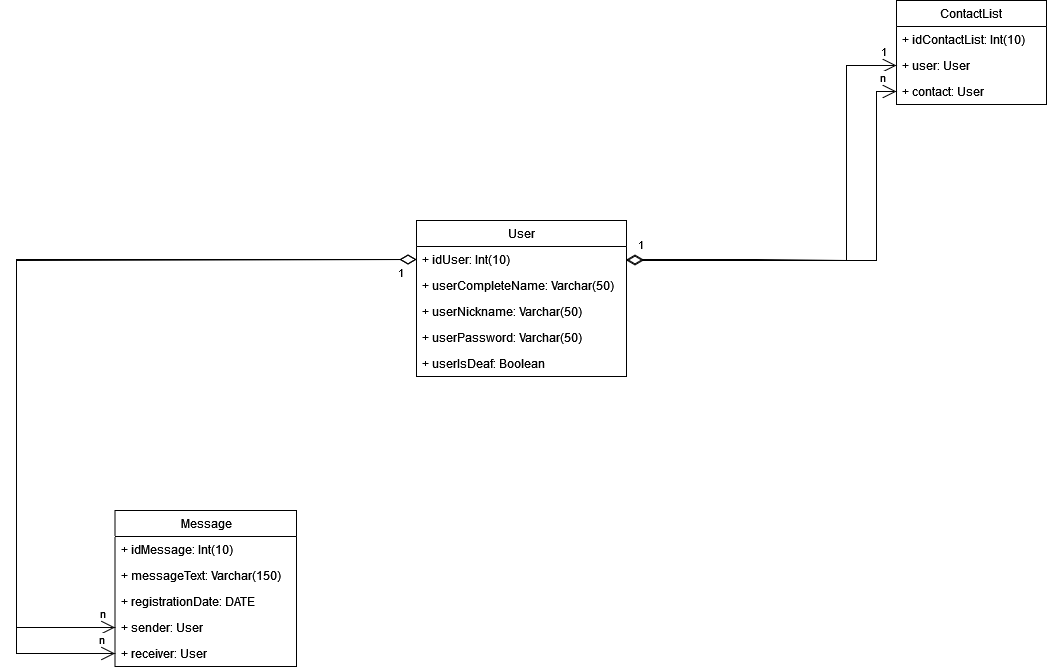


Ilustración 6. Arquitectura de la base de datos.

La arquitectura de la base de datos es muy similar a la arquitectura de la base de datos de la aplicación SignChat con ligeros cambios. La tabla usuarios ahora tiene un valor para identificar a un usuario con discapacidad. Los mensajes ahora tienen fecha de registro y la tabla de contactos fue simplificada a una sola tabla, en lugar de ser 2 tablas como solía ser antes.

### 2.3.3 Creación de diagrama de clases UML de la aplicación.

El funcionamiento es por medio de la arquitectura Modelo-Vista-Controlador o MVC. Con este modelo construimos nuestros objetos o modelos que son utilizados para la creación de estructuras en nuestra base de datos, con el controlador es donde implementamos nuestra lógica de la aplicación y en la vista implementamos lo que el usuario final vería al usar nuestro microservicio. De esta forma, el diagrama se vería de la siguiente forma:

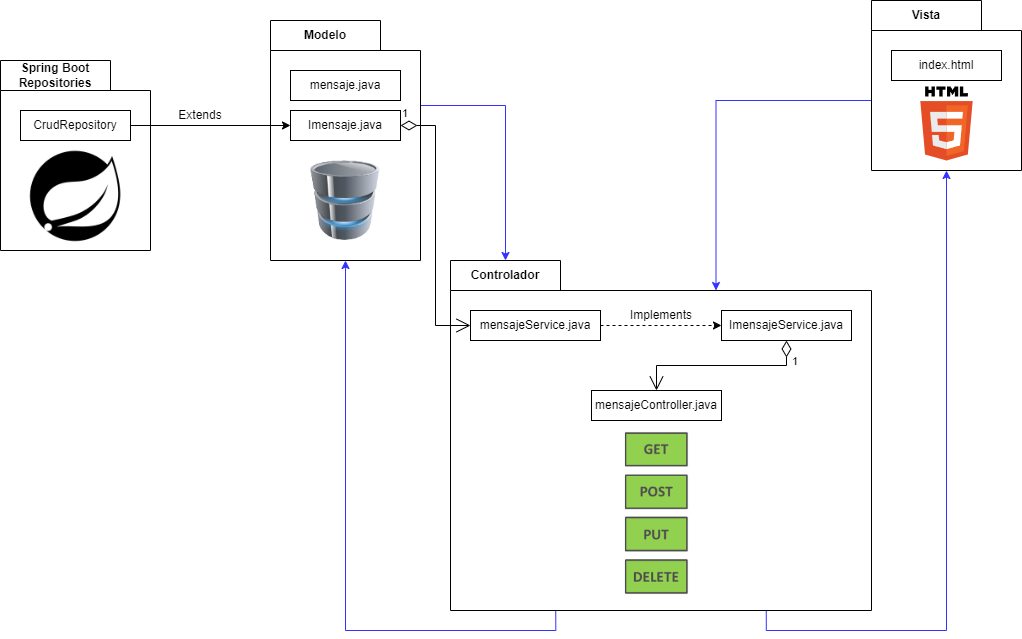


Ilustración 7.Diagrama UML de la aplicación.

En este modelo se ve representado las clases de Java y su comportamiento para el objeto mensaje. Dependiendo de la cantidad de modelos que implementemos, el modelo puede cambiar, pero para el funcionamiento principal del microservicio, solo necesitaremos del modelo mensaje.

### 2.3.4 Desarrollo de la arquitectura de integración.

Para la integración de nuestro plugin a algún proyecto, el cliente tendrá dos opciones. Su primera opción será la descarga directa del plugin, es decir el archivo JAR, el cual tendrá que ubicar en las librerías de su proyecto para que funcione. Su segunda opción es utilizar directamente el web service mediante la URL que será creada. Este web service no tiene una ubicación en específico, pero lo recomendable es siempre tener organizado estas rutas URL dentro de algún controlador del proyecto.

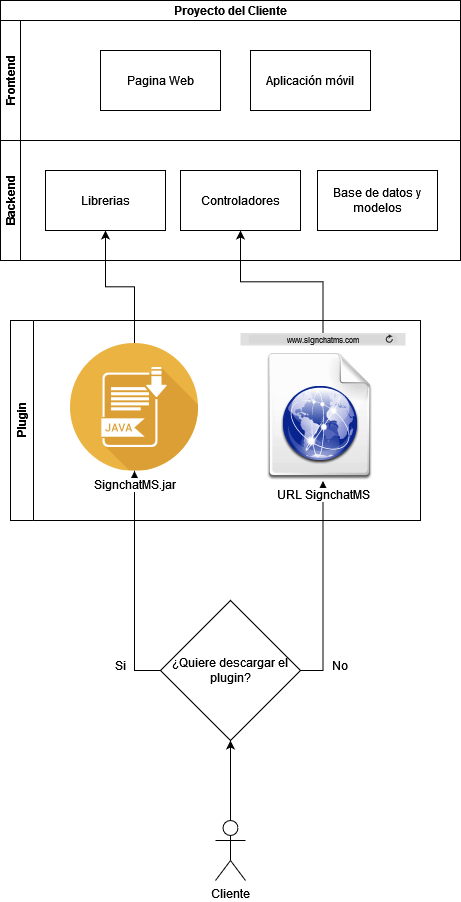


Ilustración 8. Arquitectura de Integración.

## 2.4 Desarrollo

### 2.4.1 Creación de la base de datos.

La base de datos puede ser creada automáticamente con código Java gracias al framework de Spring Boot. Utilizaremos la base de datos H2, para ello debemos implementar la siguiente dependencia en nuestro archivo pom.xml.

<dependency>

            <groupId>com.h2database</groupId>

            <artifactId>h2</artifactId>

            <version>2.2.220</version>

        </dependency>

Además de esta dependencia, cuando creamos el proyecto en Spring Boot, nos aseguramos de tener la dependencia de Spring Data JPA.

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

También utilizaremos Lombok para disminuir la cantidad de código que debemos escribir en nuestro proyecto y quede más ligero y rápido de utilizar.

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <version>1.18.20</version>

            <scope>provided</scope>

        </dependency>

Con estas dos dependencias, podemos empezar a establecer nuestra base de datos, declarando objetos dentro de los modelos establecidos. Por el momento, solo necesitamos un modelo para los mensajes y es el establecidos en Messages.java.

package com.usbbog.*S*ignchat*MS*.models;

import jakarta.persistence.\*;

import lombok.Getter;

import lombok.Setter;

import lombok.ToString;

import lombok.AllArgsConstructor;

import lombok.NoArgsConstructor;

import java.time.LocalDateTime;

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

@ToString

@Entity

@Table(name = "tbl\_Message")

public class Message {

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long idMessage;

    @Column(nullable = true)

    private String messageText;

    @Temporal(TemporalType.TIMESTAMP)

    private LocalDateTime registrationDate;

    @ManyToOne

    @JoinColumn(name = "senderIdUser")

    private User sender;

    @ManyToOne

    @JoinColumn(name = "receiverIdUser")

    private User receiver;

    public Message(String messageText) {

        this.messageText = messageText;

    }

    @PrePersist

    public void setRegistrationDate(){

        this.registrationDate = LocalDateTime.now();

    }

}

En este código utilizamos las anotaciones que nos ofrece Spring Boot y Lombok para construir la base de datos. La anotación @Entity, es para decirle a Spring Boot que cree una tabla con la estructura dada en el modelo, y con @Table le decimos que nombre queremos que tenga la tabla. Con @Getter y @Setter creamos automáticamente los getters y setters del modelo, con @ToString volvemos nuestro objeto un String para poder ser impreso por consola, que a veces nos puede resultar útil cuando debamos analizar el comportamiento de la base de datos. Por último, el @AllArgsConstructor y @NoArgsConstructor para declarar constructores con y sin parámetros. Dentro del modelo de mensajes hay una función llamada setRegistrationDate(), la cual gracias a la anotación @PrePersist, se ejecutará automáticamente cada vez que se cree un nuevo mensaje, y la función rellena automáticamente la fecha de registro con la fecha actual.

Ahora construimos el modelo de usuarios.

package com.usbbog.*S*ignchat*MS*.models;

import jakarta.persistence.\*;

import lombok.Getter;

import lombok.Setter;

import lombok.ToString;

import lombok.AllArgsConstructor;

import lombok.NoArgsConstructor;

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

@ToString

@Entity

@Table(name = "tbl\_User")

public class User{

    //Variables

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long idUser;

    @Column(nullable = false)

    private String userCompleteName;

    @Column(nullable = false, unique = true)

    private String userNickname;

    @Column(nullable = false)

    private String userPassword;

    @Column(nullable = false)

    private boolean userIsDeaf;

}

Contiene las mismas notaciones ya explicadas en el modelo mensaje.

Por último, el modelo de la lista de contactos.

package com.usbbog.*S*ignchat*MS*.models;

import jakarta.persistence.\*;

import lombok.Getter;

import lombok.Setter;

import lombok.ToString;

import lombok.AllArgsConstructor;

import lombok.NoArgsConstructor;

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

@ToString

@Entity

@Table(name = "tbl\_ContactList")

public class ContactList {

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long idContactList;

    @ManyToOne

    @JoinColumn(name = "idUser")

    private User user;

    @ManyToOne

    @JoinColumn(name = "idContact")

    private User contact;

}

### 2.4.2 Implementación de las clases establecidas en los diagramas.

Las clases establecidas en el diagrama se pueden dividir en 2 secciones: Servicios e Interfaces y Configuraciones.

#### 2.4.2.1 Servicios e Interfaces.

Las interfaces que utilizaremos son las que vienen incluidas en las herramientas Spring. Una de esas interfaces nos sirve para poder crear un Create-Read-Update-Delete de cada modelo de nuestra base de datos, también conocido como un CRUD. Un CRUD es básicamente un conjunto de funciones que nos permitirá crear, actualizar, borrar y visualizar los datos de la base de datos de la aplicación.

Conociendo esto, las interfaces de nuestros modelos se ven de la siguiente forma. Esta es la interfaz del modelo mensajes.

package com.usbbog.*S*ignchat*MS*.interfaces;

import com.usbbog.SignchatMS.models.Message;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface IMessage extends CrudRepository<Message, Long>{

}

Para la interfaz, solo heredamos los métodos del CrudRepository de Spring para poder tener el CRUD ya realizado. De la herencia, también podemos hacer consultas específicas, por ejemplo, si queremos buscar un usuario por su nombre de usuario y contraseña para una futura función de inicio de sesión, debemos implementarla en la interfaz del modelo usuario de la siguiente forma:

package com.usbbog.*S*ignchat*MS*.interfaces;

import com.usbbog.SignchatMS.models.User;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface IUser extends CrudRepository<User, Long> {

    List<User> findByUserNicknameAndUserPassword(String nickname, String password);

}

En esta interfaz creamos una función findByUserNicknameAndUserPassword(), la cual tiene ese nombre debido a la documentación de las consultas de Spring.

Por último, tenemos la interfaz de lista de contactos.

package com.usbbog.*S*ignchat*MS*.interfaces;

import com.usbbog.SignchatMS.models.ContactList;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface IContactList extends CrudRepository<ContactList, Long>{

}

Con eso, seria todas las interfaces que necesitaremos.

Ahora, debemos crear una segunda interfaz donde podamos crear las funciones CRUD que utilizaremos. Debemos crear uno por cada modelo. Esta es la interfaz del servicio para el modelo mensajes.

package com.usbbog.*S*ignchat*MS*.interface*S*ervices;

import com.usbbog.SignchatMS.models.Message;

import java.util.List;

import java.util.Optional;

public interface ImessageService {

    public List<Message> listarMensajes();

    public Optional<Message> listarMensajesId(Long id);

    public int CreateMensaje(Message p);

    public void DeleteMensaje(Long id);

}

Así es la interfaz de servicios para el modelo de usuarios.

package com.usbbog.*S*ignchat*MS*.interface*S*ervices;

import com.usbbog.SignchatMS.models.User;

import java.util.List;

import java.util.Optional;

public interface IuserService {

    public List<User> listarUsuarios();

    public Optional<User> listarUsuariosId(Long id);

    public int CreateUsuario(User p);

    public void DeleteUsuario(Long id);

}

Por último, la interfaz de servicios de la lista de contactos.

package com.usbbog.*S*ignchat*MS*.interface*S*ervices;

import com.usbbog.SignchatMS.models.ContactList;

import java.util.List;

import java.util.Optional;

public interface IcontactlistService {

    public List<ContactList> listarContactos();

    public Optional<ContactList> listarContactosId(Long id);

    public int CreateContacto(ContactList p);

    public void DeleteContacto(Long id);

}

Con esas interfaces, ya podemos utilizar los repositorios de Spring en nuestro proyecto, además de las funciones CRUD que creamos.

Para poder usar estas interfaces, debemos crear una clase Java que implemente la lógica de dichas funciones. Esta clase se conoce como servicio, y en este servicio implementamos las funciones CRUD de las interfaces y llamar el repositorio de CrudRepository de Spring.

Este es el servicio del modelo mensajes.

package com.usbbog.*S*ignchat*MS*.services;

import java.util.List;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.usbbog.SignchatMS.interfaceServices.ImessageService;

import com.usbbog.SignchatMS.interfaces.IMessage;

import com.usbbog.SignchatMS.models.Message;

@Service

public class messageService implements ImessageService {

    @Autowired

    private IMessage data;

    @Override

    public List<Message> listarMensajes(){

        return (List<Message>) data.findAll();

    }

    @Override

    public Optional<Message> listarMensajesId(Long id){

        return data.findById(id);

    }

    @Override

    public int CreateMensaje(Message p){

        int res = 0;

        Message mensaje = data.save(p);

        if(!mensaje.equals(null)){

            res = 1;

        }

        return res;

    }

    @Override

    public void DeleteMensaje(Long id){

        data.deleteById(id);

    }

}

Este es el servicio del modelo usuarios.

package com.usbbog.*S*ignchat*MS*.services;

import com.usbbog.SignchatMS.interfaceServices.IuserService;

import com.usbbog.SignchatMS.interfaces.IUser;

import com.usbbog.SignchatMS.models.User;

import java.util.List;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class userService implements IuserService{

    @Autowired

    private IUser data;

    @Override

    public List<User> listarUsuarios(){

        return (List<User>) data.findAll();

    }

    @Override

    public Optional<User> listarUsuariosId(Long id){

        return data.findById(id);

    }

    @Override

    public int CreateUsuario(User p){

        int res = 0;

        User usuario = data.save(p);

        if(!usuario.equals(null)){

            res = 1;

        }

        return res;

    }

    @Override

    public void DeleteUsuario(Long id){

        data.deleteById(id);

    }

}

Este es el servicio del modelo de la lista de contactos.

package com.usbbog.*S*ignchat*MS*.services;

import java.util.List;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.usbbog.SignchatMS.interfaceServices.IcontactlistService;

import com.usbbog.SignchatMS.interfaces.IContactList;

import com.usbbog.SignchatMS.models.ContactList;

@Service

public class contactlistService implements IcontactlistService {

    @Autowired

    private IContactList data;

    @Override

    public List<ContactList> listarContactos(){

        return (List<ContactList>) data.findAll();

    }

    @Override

    public Optional<ContactList> listarContactosId(Long id){

        return data.findById(id);

    }

    @Override

    public int CreateContacto(ContactList p){

        int res = 0;

        ContactList contacto = data.save(p);

        if(!contacto.equals(null)){

            res = 1;

        }

        return res;

    }

    @Override

    public void DeleteContacto(Long id){

        data.deleteById(id);

    }

}

#### 2.4.2.2 Configuraciones.

Cuando una aplicación web o móvil utiliza alguna API externa que no está en su dominio, se puede presentar la falla que genera CORS. CORS es una implementación de seguridad que evita principalmente que páginas webs externas accedan a las funcionalidades de otras páginas web. Para nuestro microservicio, debemos configurar CORS para que cualquier aplicación pueda acceder a nuestras funcionalidades. Dicha configuración de CORS es la siguiente:

package com.usbbog.*S*ignchat*MS*.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.CorsRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

public class CorsConfig implements WebMvcConfigurer {

    @Override

    public void addCorsMappings(CorsRegistry registry){

        registry.addMapping("/\*\*")

                .allowedOriginPatterns("\*")

                .allowedMethods("GET", "POST", "PUT", "DELETE")

                .allowedHeaders("\*")

                .allowCredentials(true);

    }

}

Utilizando “\*” le decimos a nuestra aplicación Spring Boot que cualquier aplicación puede utilizar nuestro proyecto.

Además de CORS, debemos configurar nuestro servidor Websocket. Debemos configurar una ruta donde llegaran los mensajes y la ruta principal del websocket, además de decirle que tecnología vamos a utilizar, que en nuestro caso será SockJS. La configuración del Websocket es la siguiente.

package com.usbbog.*S*ignchat*MS*.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.messaging.simp.config.MessageBrokerRegistry;

import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;

import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override

    public void configureMessageBroker(MessageBrokerRegistry registry) {

        registry.enableSimpleBroker("/topic");

        registry.setApplicationDestinationPrefixes("/app");

    }

    @Override

    public void registerStompEndpoints(StompEndpointRegistry registry) {

        registry

                .addEndpoint("/ws-chat")

                .setAllowedOriginPatterns("\*")

                .withSockJS();

    }

}

Al igual que en CORS, utilizamos “\*” para que el websocket pueda ser accedido por cualquier aplicación para poder usar los mensajes en tiempo real.

### 2.4.3 Codificación de las funcionalidades requeridas.

Para las funcionalidades, debemos crear los controladores de cada modelo y de otros requerimientos de la aplicación. Los controladores por lo general contienen las rutas que los clientes llamaran para poder utilizar el microservicio.

#### 2.4.3.1 Controlador de lista de contactos.

Las rutas para el modelo de la lista de contactos son pocas. Se construyo la función para crear un contacto nuevo y poder ver dicha lista de contactos. El controlador quedo de la siguiente forma.

package com.usbbog.*S*ignchat*MS*.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

import com.usbbog.SignchatMS.interfaceServices.IcontactlistService;

import com.usbbog.SignchatMS.models.ContactList;

@RestController

@RequestMapping("/api/contacts/")

public class ContactlistController {

    @Autowired

    private IcontactlistService service;

    @PostMapping("/create")

    public String crearContacto(Model model, ContactList contacto){

        service.CreateContacto(contacto);

        return "Contact Created";

    }

    @GetMapping("")

    public List<ContactList> mostrarContactos(){

        List<ContactList> contactos = service.listarContactos();

        return contactos;

    }

}

Establecemos que es una API REST utilizando la anotación @RestController. La anotación @RequestMapping nos ayudara a definir la ruta para todos los métodos que creemos en el controlador. Después creamos las funciones y llamamos el servicio del modelo creado. Cada función tiene una anotación, que puede ser @PostMapping o @GetMapping, la cual depende si vamos a utilizar un método POST o GET junto a su ruta para llamar dicha función.

#### 2.4.3.2 Controlador de usuarios.

El controlador de usuarios contiene también un método para crear y visualizar un usuario, pero también tiene una función para hacer inicio de sesión. Esta función fue implementada en caso a que el cliente requiera utilizar sesiones dentro de su aplicación.

package com.usbbog.*S*ignchat*MS*.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

import com.usbbog.SignchatMS.interfaceServices.IuserService;

import com.usbbog.SignchatMS.interfaces.IUser;

import com.usbbog.SignchatMS.models.User;

@RestController

@RequestMapping("/api/users/")

public class UserController {

    @Autowired

    private IuserService service;

    @Autowired

    private IUser query;

    @PostMapping("/create")

    public String crearUsuario(Model model, User usuario){

        service.CreateUsuario(usuario);

        return "User created";

    }

    @GetMapping("")

    public List<User> mostrarUsuarios(){

        List<User> usuarios = service.listarUsuarios();

        return usuarios;

    }

    @PostMapping("/login")

    public String LoginUsuario(@RequestParam String nickname, @RequestParam String password){

        System.out.println("Nickname: " + nickname);

        System.out.println("Password: " + password);

        List<User> usuarios = query.findByUserNicknameAndUserPassword(nickname, password);

        for (User usuario : usuarios) {

            System.out.println(usuario);

        }

        return "";

    }

}

#### 2.4.3.3 Controlador de mensajes.

Dentro del controlador de mensajes es donde se encuentra toda la lógica para la traducción de mensajes. Además de los métodos de crear y visualizar mensaje, también están las funciones para traducir los mensajes de texto a imagen y viceversa.

package com.usbbog.*S*ignchat*MS*.controllers;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

import com.usbbog.SignchatMS.interfaceServices.ImessageService;

import com.usbbog.SignchatMS.models.Message;

import jakarta.annotation.PostConstruct;

@RestController

@RequestMapping("/api/messages/")

public class MessageController {

    //Traductores

    private final Map<String, String> traductorImgTxt = new HashMap<>();

    private final Map<String, String> traductorTxtImg = new HashMap<>();

    @Value("${image.baseurl}")

    private String imageBaseUrl;

    @PostConstruct

    public void init() {

        //Traductor de imagenes a texto

        traductorImgTxt.put(imageBaseUrl + "sc\_0.png", "0");

        traductorImgTxt.put(imageBaseUrl + "sc\_1.png", "1");

        traductorImgTxt.put(imageBaseUrl + "sc\_2.png", "2");

        traductorImgTxt.put(imageBaseUrl + "sc\_3.png", "3");

        traductorImgTxt.put(imageBaseUrl + "sc\_4.png", "4");

        traductorImgTxt.put(imageBaseUrl + "sc\_5.png", "5");

        traductorImgTxt.put(imageBaseUrl + "sc\_6.png", "6");

        traductorImgTxt.put(imageBaseUrl + "sc\_7.png", "7");

        traductorImgTxt.put(imageBaseUrl + "sc\_8.png", "8");

        traductorImgTxt.put(imageBaseUrl + "sc\_9.png", "9");

        traductorImgTxt.put(imageBaseUrl + "sc\_a.png", "a");

        traductorImgTxt.put(imageBaseUrl + "sc\_b.png", "b");

        traductorImgTxt.put(imageBaseUrl + "sc\_c.png", "c");

        traductorImgTxt.put(imageBaseUrl + "sc\_d.png", "d");

        traductorImgTxt.put(imageBaseUrl + "sc\_e.png", "e");

        traductorImgTxt.put(imageBaseUrl + "sc\_f.png", "f");

        traductorImgTxt.put(imageBaseUrl + "sc\_g.png", "g");

        traductorImgTxt.put(imageBaseUrl + "sc\_h.png", "h");

        traductorImgTxt.put(imageBaseUrl + "sc\_i.png", "i");

        traductorImgTxt.put(imageBaseUrl + "sc\_j.png", "j");

        traductorImgTxt.put(imageBaseUrl + "sc\_k.png", "k");

        traductorImgTxt.put(imageBaseUrl + "sc\_l.png", "l");

        traductorImgTxt.put(imageBaseUrl + "sc\_m.png", "m");

        traductorImgTxt.put(imageBaseUrl + "sc\_n.png", "n");

        traductorImgTxt.put(imageBaseUrl + "sc\_ni.png", "ñ");

        traductorImgTxt.put(imageBaseUrl + "sc\_o.png", "o");

        traductorImgTxt.put(imageBaseUrl + "sc\_p.png", "p");

        traductorImgTxt.put(imageBaseUrl + "sc\_q.png", "q");

        traductorImgTxt.put(imageBaseUrl + "sc\_r.png", "r");

        traductorImgTxt.put(imageBaseUrl + "sc\_s.png", "s");

        traductorImgTxt.put(imageBaseUrl + "sc\_t.png", "t");

        traductorImgTxt.put(imageBaseUrl + "sc\_u.png", "u");

        traductorImgTxt.put(imageBaseUrl + "sc\_v.png", "v");

        traductorImgTxt.put(imageBaseUrl + "sc\_w.png", "w");

        traductorImgTxt.put(imageBaseUrl + "sc\_x.png", "x");

        traductorImgTxt.put(imageBaseUrl + "sc\_y.png", "y");

        traductorImgTxt.put(imageBaseUrl + "sc\_z.png", "z");

        //traductorImgTxt.put(imageBaseUrl + "space\_view.png", " ");

        traductorImgTxt.put(" ", " ");

        //Traductor de texto a imagenes

        traductorTxtImg.put("0", imageBaseUrl + "sc\_0.png");

        traductorTxtImg.put("1", imageBaseUrl + "sc\_1.png");

        traductorTxtImg.put("2", imageBaseUrl + "sc\_2.png");

        traductorTxtImg.put("3", imageBaseUrl + "sc\_3.png");

        traductorTxtImg.put("4", imageBaseUrl + "sc\_4.png");

        traductorTxtImg.put("5", imageBaseUrl + "sc\_5.png");

        traductorTxtImg.put("6", imageBaseUrl + "sc\_6.png");

        traductorTxtImg.put("7", imageBaseUrl + "sc\_7.png");

        traductorTxtImg.put("8", imageBaseUrl + "sc\_8.png");

        traductorTxtImg.put("9", imageBaseUrl + "sc\_9.png");

        traductorTxtImg.put("a", imageBaseUrl + "sc\_a.png");

        traductorTxtImg.put("b", imageBaseUrl + "sc\_b.png");

        traductorTxtImg.put("c", imageBaseUrl + "sc\_c.png");

        traductorTxtImg.put("d", imageBaseUrl + "sc\_d.png");

        traductorTxtImg.put("e", imageBaseUrl + "sc\_e.png");

        traductorTxtImg.put("f", imageBaseUrl + "sc\_f.png");

        traductorTxtImg.put("g", imageBaseUrl + "sc\_g.png");

        traductorTxtImg.put("h", imageBaseUrl + "sc\_h.png");

        traductorTxtImg.put("i", imageBaseUrl + "sc\_i.png");

        traductorTxtImg.put("j", imageBaseUrl + "sc\_j.png");

        traductorTxtImg.put("k", imageBaseUrl + "sc\_k.png");

        traductorTxtImg.put("l", imageBaseUrl + "sc\_l.png");

        traductorTxtImg.put("m", imageBaseUrl + "sc\_m.png");

        traductorTxtImg.put("n", imageBaseUrl + "sc\_n.png");

        traductorTxtImg.put("ñ", imageBaseUrl + "sc\_ni.png");

        traductorTxtImg.put("o", imageBaseUrl + "sc\_o.png");

        traductorTxtImg.put("p", imageBaseUrl + "sc\_p.png");

        traductorTxtImg.put("q", imageBaseUrl + "sc\_q.png");

        traductorTxtImg.put("r", imageBaseUrl + "sc\_r.png");

        traductorTxtImg.put("s", imageBaseUrl + "sc\_s.png");

        traductorTxtImg.put("t", imageBaseUrl + "sc\_t.png");

        traductorTxtImg.put("u", imageBaseUrl + "sc\_u.png");

        traductorTxtImg.put("v", imageBaseUrl + "sc\_v.png");

        traductorTxtImg.put("w", imageBaseUrl + "sc\_w.png");

        traductorTxtImg.put("x", imageBaseUrl + "sc\_x.png");

        traductorTxtImg.put("y", imageBaseUrl + "sc\_y.png");

        traductorTxtImg.put("z", imageBaseUrl + "sc\_z.png");

        traductorTxtImg.put(" ", imageBaseUrl + "space\_view.png");

    }

    @Autowired

    private ImessageService service;

    @PostMapping("/create")

    public String crearMensaje(Model model, Message mensaje){

        service.CreateMensaje(mensaje);

        return "Message Created";

    }

    @GetMapping("")

    public List<Message> mostrarMensajes(){

        List<Message> mensajes = service.listarMensajes();

        return mensajes;

    }

    @PostMapping("/translate/img-txt")

    public String traducirImagenes(@RequestBody Map<String, Object> body){

        List<String> imagenes = new ArrayList<>();

        //Pasamos el body a una lista

        for(Object value : body.values()){

            if (value instanceof List){

                List<String> lista = (List<String>) value;

                imagenes.addAll(lista);

            }

        }

        //Traducimos el mensaje

        StringBuilder mensaje = new StringBuilder();

        for (String imagen : imagenes) {

            String significado = traductorImgTxt.get(imagen);

            if (significado != null){

                mensaje.append(significado);

            }

        }

        return mensaje.toString();

    }

    @PostMapping("/translate/txt-img")

    public List<String> traducirTextoAImagenes(@RequestBody Map<String, String> body) {

        String texto = body.get("text");

        List<String> imagenes = new ArrayList<>();

        // Iterar a través del texto y buscar la ruta de imagen correspondiente para cada letra

        for (char letra : texto.toCharArray()) {

            String rutaImagen = traductorTxtImg.get(String.valueOf(letra));

            if (rutaImagen != null) {

                imagenes.add(rutaImagen);

            }

        }

        return imagenes;

    }

}

Para la traducción de los mensajes utilizamos un hashmap. Este hashmap guarda el valor de la imagen, según su ruta para llamar la imagen desde la aplicación, en la letra que representa. También se creó un segundo hashmap, pero con los valores inversos para poder traducir de las dos formas sin complicaciones. Los hashmaps están dentro del método init(), el cual se inicia después de construir la aplicación de Spring Boot como afirma la anotación @PostConstruct.

#### 2.4.3.4 Controlador del teclado LSC.

Vimos que, en controlador de los mensajes, los hashmaps guardan el valor de la imagen dependiendo de una ruta de la aplicación Spring Boot para identificar la imagen, dicha ruta se encuentra en el controlador del teclado.

package com.usbbog.*S*ignchat*MS*.controllers;

import jakarta.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.core.io.Resource;

import org.springframework.core.io.ResourceLoader;

import org.springframework.http.ResponseEntity;

import org.springframework.util.StreamUtils;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RestController;

import java.io.IOException;

import java.io.InputStream;

@RestController

public class KeyboardController {

    private final ResourceLoader resourceLoader;

    @Value("${image.baseurl}")

    private String imageBaseUrl;

    public KeyboardController(ResourceLoader resourceLoader) {

        this.resourceLoader = resourceLoader;

    }

    @GetMapping("/images/{imageName}")

    public void getImage(@PathVariable String imageName, HttpServletResponse response) throws IOException {

        Resource resource = resourceLoader.getResource("classpath:/static/keyboard/" + imageName);

        if (resource.exists()){

            InputStream inputStream = resource.getInputStream();

            response.setContentType("image/jpeg"); //Adaptar el tipo de imagen (PNG, JPEG, etc)

            StreamUtils.copy(inputStream, response.getOutputStream());

        } else {

            // TODO: Hacer algo en caso de que no exista la imagen

        }

    }

    @GetMapping("/keyboard")

    public ResponseEntity<String> getSpecialKeyboard(){

        StringBuilder html = new StringBuilder();

        html.append("<div class=\"keyboard\">");

        //Generar los <img/> para cada tecla 0-9

        for(char key = '0'; key <= '9'; key++){

            String imageName = "sc\_" + key + ".png";

            html.append("<button class=\"key-button\" data-key=\"").append(key).append("\" onclick=\"LscKeyPressed(this)\">");

            html.append("<img src=\"").append(imageBaseUrl).append(imageName).append("\" alt=\"").append(key).append("\">");

            html.append("</button>");

        }

        html.append("<br />");

        //Generar los <img/> para cada tecla a-z

        for(char key = 'a'; key <= 'z'; key++){

            String imageName = "sc\_" + key + ".png";

            html.append("<button class=\"key-button\" data-key=\"").append(key).append("\" onclick=\"LscKeyPressed(this)\">");

            html.append("<img src=\"").append(imageBaseUrl).append(imageName).append("\" alt=\"").append(key).append("\">");

            html.append("</button>");

            //Letra ñ

            if (key == 'n'){

                html.append("<button class=\"key-button\" data-key=\"").append("ñ").append("\" onclick=\"LscKeyPressed(this)\">");

                html.append("<img src=\"").append(imageBaseUrl).append("sc\_ni.png").append("\" alt=\"").append("ñ").append("\">");

                html.append("</button>");

            }

            //Salto de Linea cada 10 teclas

            if ((key - 'a' + 1) % 10 == 0){

                html.append("<br />");

            }

        }

        html.append("<br />");

        //Teclas SPACE y BORRAR

        html.append("<button class=\"key-button\" data-key=\"").append(" ").append("\" onclick=\"LscKeyPressed(this)\">");

        html.append("<img src=\"").append(imageBaseUrl).append("space\_key.png").append("\" alt=\"").append(" ").append("\">");

        html.append("</button>");

        html.append("<button class=\"key-button\" data-key=\"").append("del").append("\" onclick=\"LscKeyPressed(this)\">");

        html.append("<img src=\"").append(imageBaseUrl).append("delete\_key.png").append("\" alt=\"").append("del").append("\">");

        html.append("</button>");

        // Agregar el script para manejar el evento de clic

        html.append("<script>");

        html.append("function LscKeyPressed(button) {");

        html.append("    var key = button.getAttribute('data-key');");

        html.append("    console.log('Tecla clickeada:', key);");

        html.append("}");

        html.append("</script>");

        html.append("</div>");

        return ResponseEntity.ok().body(html.toString());

    }

    @GetMapping("/keyboard/react")

    public ResponseEntity<String> getSpecialKeyboardReact(){

        StringBuilder html = new StringBuilder();

        html.append("<div class=\"keyboard\">");

        //Generar los <img/> para cada tecla 0-9

        for(char key = '0'; key <= '9'; key++){

            String imageName = "sc\_" + key + ".png";

            //html.append("<button className=\"key-button\" data-key=\"").append(key).append("\" onClick={handleButtonClick}>");

            html.append("<button className=\"key-button\" data-key=\"").append(key).append("\">");

            //html.append("<button className=\"key-button\" data-key=\"").append(key).append("\" onclick=\"handleButtonClick(this)\">");

            html.append("<img src=\"").append(imageBaseUrl).append(imageName).append("\" alt=\"").append(key).append("\">");

            html.append("</button>");

        }

        html.append("<br />");

        //Generar los <img/> para cada tecla a-z

        for(char key = 'a'; key <= 'z'; key++){

            String imageName = "sc\_" + key + ".png";

            html.append("<button className=\"key-button\" data-key=\"").append(key).append("\">");

            //html.append("<button className=\"key-button\" data-key=\"").append(key).append("\" onClick={handleButtonClick}>");

            //html.append("<button className=\"key-button\" data-key=\"").append(key).append("\" onclick=\"handleButtonClick(this)\">");

            html.append("<img src=\"").append(imageBaseUrl).append(imageName).append("\" alt=\"").append(key).append("\">");

            html.append("</button>");

            //Letra ñ

            if (key == 'n'){

                html.append("<button class=\"key-button\" data-key=\"").append("ñ").append("\">");

                html.append("<img src=\"").append(imageBaseUrl).append("sc\_ni.png").append("\" alt=\"").append("ñ").append("\">");

                html.append("</button>");

            }

            //Salto de Linea cada 10 teclas

            if ((key - 'a' + 1) % 10 == 0){

                html.append("<br />");

            }

        }

        html.append("<br />");

        //Teclas SPACE y BORRAR

        String spaceImageAlt = imageBaseUrl + "space\_view.png";

        html.append("<button class=\"key-button\" data-key=\"").append(" ").append("\">");

        html.append("<img src=\"").append(imageBaseUrl).append("space\_key.png").append("\" alt=\"").append(" ").append("\">");

        html.append("</button>");

        html.append("<button class=\"key-button\" data-key=\"").append("del").append("\">");

        html.append("<img src=\"").append(imageBaseUrl).append("delete\_key.png").append("\" alt=\"").append("del").append("\">");

        html.append("</button>");

        // Agregar el script para manejar el evento de clic

//        html.append("<script>");

//        html.append("function LscKeyPressed(button) {");

//        html.append("    var key = button.getAttribute('data-key');");

//        html.append("    console.log('Tecla clickeada:', key);");

//        html.append("}");

//        html.append("</script>");

//

//        html.append("</div>");

        return ResponseEntity.ok().body(html.toString());

    }

}

En este controlador hay 2 funciones importantes: la función para llamar una imagen única y la función para construir el teclado utilizando la primera función. Las imágenes ya están cargadas y almacenadas dentro de la aplicación de Spring Boot, por lo que solo necesitamos utilizar un controlador para mostrar las imágenes. El problema es que solo podemos mostrar una imagen a la vez y necesitamos mostrar todas las imágenes. Lo que realiza la segunda función, utilizando un StringBuilder, construimos un elemento HTML que se asemeje lo mejor posible a un teclado, llamando todas las imágenes del microservicio para su construcción, además de incluir un pequeño script de JavaScript para comprobar su funcionamiento.

Podemos notar también que la función para construir el teclado se repite, esto debido a que, en la actualidad, hay varias tecnologías para crear aplicaciones web. Una de las tecnologías más comunes es ReactJS, por lo que la segunda función del teclado, aunque similar en su mayoría, fue creada para que una página web que utilice tecnologías muy modernas también pueda acceder al teclado.

#### 2.4.3.5 Controlador del websocket.

En este controlador encontraremos el comportamiento del websocket.

package com.usbbog.*S*ignchat*MS*.controllers;

import com.usbbog.SignchatMS.models.Message;

import org.apache.coyote.Response;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.HttpStatusCode;

import org.springframework.http.ResponseEntity;

import org.springframework.messaging.handler.annotation.MessageMapping;

import org.springframework.messaging.handler.annotation.Payload;

import org.springframework.messaging.handler.annotation.SendTo;

import org.springframework.messaging.simp.SimpMessagingTemplate;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.util.HtmlUtils;

@RestController

public class WebSocketController {

    @Autowired

    SimpMessagingTemplate template;

    @PostMapping("/send")

    public ResponseEntity<Void> sendMessage(@RequestBody Message mensaje){

        template.convertAndSend("/topic/message", mensaje);

        return new ResponseEntity<>(HttpStatus.OK);

    }

    @MessageMapping("/sendMessage")

    public void receiveMessage(@Payload Message mensaje){

        //TODO: Manejar el recibir mensaje ws

    }

    @SendTo("/topic/message")

    public Message broadcastMessage(@Payload Message mensaje){

        return mensaje;

    }

}

El Websocket contiene 3 funciones, aunque solo utiliza 2 de ellas. Tiene la función para lo que tiene que hacer cuando se envía el mensaje y para transmitir el mensaje en el servidor

### 2.4.4 Creación de páginas web de prueba.

#### 2.4.4.1 Php

Para la implementación del teclado virtual, se desarrolló una versión de una página HTML compuesta de tres partes un área de texto no editable donde se publica el texto, un cuadro de texto para ingresar la información ya sea directamente o utilizando el teclado virtual, y por último un espacio done se trae y publica el teclado virtual desarrollado en Spring Boot. Este teclado se obtiene mediante la combinación de los lenguajes Php y HTML. Se debe especificar que para el manejo de funciones utilizadas por los botones debe ser implementado otro lenguaje de manipulación del lado del cliente.

En el siguiente código se evidencia el contenedor del teclado virtual y cómo se realiza la solicitud al URL y su respectiva extensión para traer el teclado virtual utilizando *file\_get\_contents* y la ruta */keyboard*.

|  |
| --- |
| <div class="input-field" id="keyboard-container">  <?php  //Obtener el contenido de "<http://localhost:8080/keyboard>" utilizando PHP  $contenido = file\_get\_contents('http://localhost:8080/keyboard');  // Enseñar el contenido dentro de la página  echo $contenido;  ?>  </div> |

Otra forma de solicitar el teclado virtual se puede realizar utilizando el elemento *iframe* de HTML convidando con la dirección del teclado virtual.

|  |
| --- |
| <iframe src="http://localhost:8080/keyboard" frameborder="0"></iframe> |

El espacio donde se publica el resultado se ve de la siguiente manera:

|  |
| --- |
| HTML implementa teclado  Ilustración 9. Teclado de señas virtual |

#### 2.4.4.2 JavaScript

En este apartado se busca el ilustrar y explicar la forma en que se realiza la implementación y uso del programa desarrollado y su comportamiento al ser implementado con el lenguaje JavaScript. Para enseñarlo se ha creado una página HTML básica para visualizar la implementación.

Para cargar el contenido del teclado de lenguaje de señas se debe realizar la solicitud del teclado, lo cual se hace al solicitar a una dirección URL donde se ejecuta Spring Boot seguido de la extensión que diseña la estructura del teclado.

Imagen del teclado virtual desarrollado en Spring Boot.

|  |
| --- |
| HTML implementa teclado  Ilustración 10. Teclado de señas virtual |

Función JavaScript para la carga del contenido del teclado.

|  |
| --- |
| function cargarContenido() {  // Realizar una solicitud HTTP GET a la URL que alberga el teclado  fetch('http://localhost:8080/keyboard')  .then(response => response.text())  .then(data => {  document.getElementById('keyboard-container').innerHTML = data;  connect();  })  .catch(error => console.error('Error:', error));  } |

La dirección *http://localhost:8080/keyboard* llama la estructura del teclado, y se publica en la página desarrollada. La función utilizada para cargar y publicar el teclado, ”*cargarContenido()”* implementa otra función llamada “*connect()”* que cumple la función de establecer una conexión con la base de datos temporal creada por Spring Boot para poder manejar websockets.

|  |
| --- |
| *var* stompClient = null;  *function* connect() {  console.log(typeof SockJS);  *//URL para conección al endpoint WebSocket*  *var* socket = new SockJS('http://localhost:8080/ws-chat');  stompClient = Stomp.over(socket);  stompClient.connect({}, *function* (*frame*) {  console.log('Conectado: ' + frame);  });  *// Suscribirse al canal después de la conexión exitosa*  stompClient.subscribe('/topic/message', *function* (*response*) {  *var* message = JSON.parse(response.body);  *// Manejar el mensaje recibido y mostrarlo en el chat-container*  *var* chatContainer = document.getElementById('chat-container');  chatContainer.innerHTML += '<p>' + message.message + '</p>';  });  }  // dependencias necesarias para utilizar SockJS y Stopm     <script *src*="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.6.1/sockjs.min.js" *integrity*="sha512-1QvjE7BtotQjkq8PxLeF6P46gEpBRXuskzIVgjFpekzFVF4yjRgrQvTG1MTOJ3yQgvTteKAcO7DSZI92+u/yZw==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script>      <script *src*="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.6.1/sockjs.js" *integrity*="sha512-aHuAnb+FZ/yM5zYvdiA8q0sxLdsBQ3PRwO8D/08qsenCLQ2gWsosB43Cn1X8esAz0IpaC7Ns5cAPyKIdJQ0hNw==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script>      <script *src*="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js" *integrity*="sha512-iKDtgDyTHjAitUDdLljGhenhPwrbBfqTKWO1mkhSFH3A7blITC9MhYon6SjnMhp4o0rADGw9yAC6EW4t5a4K3g==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script>      <script *src*="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.js" *integrity*="sha512-tL4PIUsPy+Rks1go4kQG8M8/ItpRMvKnbBjQm4d2DQnFwgcBYRRN00QdyQnWSCwNMsoY/MfJY8nHp2CzlNdtZA==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script> |

Estas funciones se utilizan para establecer una conexión con el servidor temporal creado con Spring Boot y poder implementar el procedimiento de web service creado para la comunicación en tiempo real utilizando la variable *stompClient*. Para el manejo de la conexión a websocket se crea un objeto StompJS bajo el nombre de socket para representar su conexión (valor que se le asigna a la variable de manejo).

En la segunda parte del código se trata de la suscripción al canal */topic/message* del servidor creado al ejecutar Spring Boot donde se hizo la conexión publicando un mensaje de conexión exitosa en la consola, además de publicar el mensaje en el contenedor HTML.

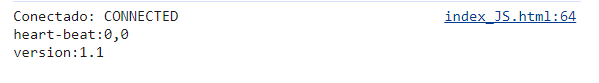


Ilustración 11. Mensaje de conexión exitoso

Los botones del teclado tienen una función donde al ser seleccionados se ejecuta una función que publica el valor del botón elegido en un cuadro de texto.

|  |
| --- |
| function LscKeyPressed(button) {  var dataKey = button.getAttribute('data-key');  var messageInput = document.getElementById('message-input');    // Verificar si el boton elegido es BORRAR se elimina el último carácter de la secuencia, en caso contrario se agrega en la sentencia  if (dataKey === "del") {  var inputValue = messageInput.value;  if (inputValue.length > 0) {  // Eliminar el último carácter  messageInput.value = inputValue.slice(0, -1);  }  } else {  // Agregar el valor de dataKey al input-field  messageInput.value += dataKey;  }  } |

En la siguiente imagen se muestra la página HTML de ejemplo que implementa el teclado virtual utilizando las funciones anteriores publicando el valor de los botones en el cuadro de texto.

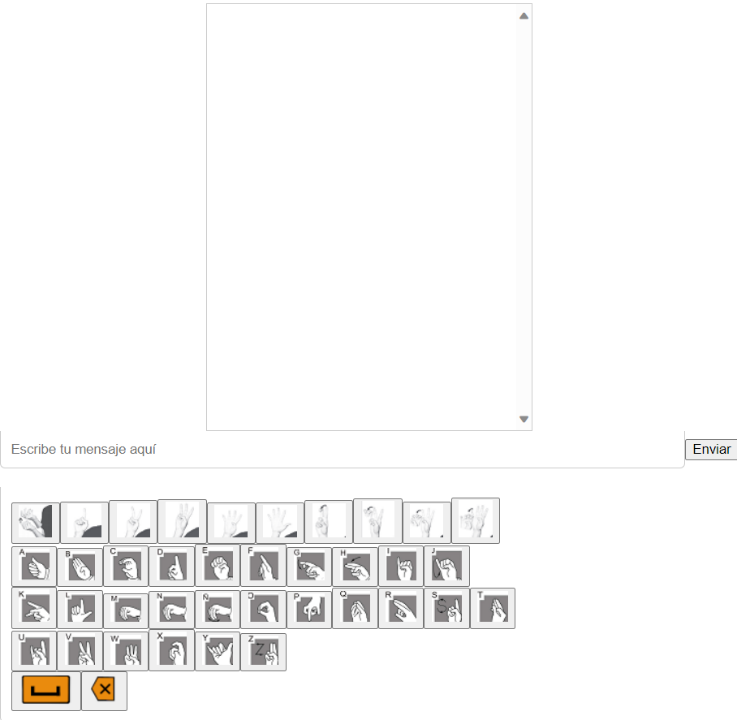


Ilustración 12. Pagina HTML que implementa teclado mediante JavaScript

En situaciones donde hay más input se desarrolló otra página HTML que simula el inicio de sesión, donde existen diferentes cantidades de inputs donde dependiendo el elegido será en el que se publicarían las elecciones del teclado virtual.

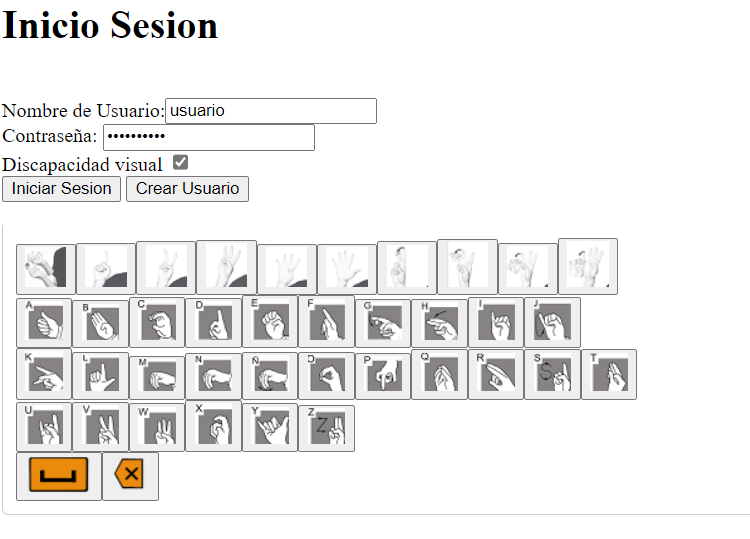


Ilustración 13. Pagina HTML #2 que implementa teclado usando JavaScript

Para establecer una distinción entre los cuadros de texto donde se publicará el resultado de las elecciones del teclado se implementan una serie de funciones para realizar esta distinción.

|  |
| --- |
| *//Los inputs deben implementar la función SelectInput enviando el valor de identidad del mismo*  *onclick*="SelecionInput(*this*)" |

La función utiliza un array donde se almacena la identidad del input para ser utilizada en otras operaciones y se verifica que cumpla las condiciones de llenar la opción CheckBox para solicitar el teclado virtual.

|  |
| --- |
| *var* inputIDs = []; *// Array para almacenar las ID de los inputs*  *function* SelecionInput(*inputElement*){  *// Obtén el valor del id del input*  *var* idInput = inputElement.id;                  inputIDs.splice(0, inputIDs.length);                  inputIDs.push(idInput);                  console.log("input:: "+ inputIDs);  *// Obtén el estado del checkbox*  *var* checkbox = document.getElementById("discapacidad");  *var* divKeyB = document.getElementById("keyboard-container");                  if (checkbox.checked) {  *// Si el checkbox está seleccionado, muestra el teclado*                      console.log('check');                      divKeyB.style.visibility ="visible";                  } else {  *// Si el checkbox no está seleccionado, oculta el teclado*                      console.log('not check');                      divKeyB.style.visibility ="hidden";                  }              } |

Para realizar una diferenciación entre los inputs se utiliza el arreglo *inputIDs* que almacena la identidad del input actual y se actualiza o borra conforme se elija otro input. Se utiliza la función *LscKeyPressed* donde una variable toma el valor almacenado en el arreglo para ser utilizado para identificar el destino de las elecciones echas con el teclado.

|  |
| --- |
| *function* LscKeyPressed(*button*) {  *var* dataKey = button.getAttribute('data-key');  *var* messageInput = document.getElementById(inputIDs);              console.log("input:: "+ inputIDs);  *// Verificar si dataKey es "del" y eliminar el último carácter*              if (dataKey === "del") {  *var* inputValue = messageInput.value;                  if (inputValue.length > 0) {  *// Eliminar el último carácter*                      messageInput.value = inputValue.slice(0, -1);                  }              } else {  *// Agregar el valor de dataKey al input-field*                  messageInput.value += dataKey;              }              } |

### 2.4.5 Retroalimentación y corrección de errores.

Durante el desarrollo del proyecto se han identificado y solucionado situaciones que han aparecido en este periodo.

#### 2.4.5.1 Problema en la configuración de CORS

Uno de los problemas más grandes del proyecto fue la conectividad del microservicio. Cuando buscamos que el microservicio este apto para cualquier página web, debemos abrir los puertos a todo el mundo, lo cual es peligroso porque puede resultar que quien utilizo el microservicio fue para dañarlo y hacer un ciberataque al sistema.

Para solucionar este problema, le especificamos a CORS que rutas de la API REST son aptos para todo el mundo y que métodos de REST pueden realizar. Esto genero problemas porque hay páginas web que no lograban conectar al microservicio, o no podían acceder por completo a su funcionalidad.

#### 2.4.5.2 Problema en la conexión con los websockets

La implementación de los websockets tuvo ciertos obstáculos. El mayor de esos obstáculos era el habilitar los permisos, similar a CORS, para que el publico pueda acceder a él. Hubo problema porque al hacer esto, sobrepasa las normas que habíamos establecido en CORS y debíamos, de alguna forma, balancear ambas políticas de ambas tecnologías para que pudieran funcionar de manera síncrona y sin problemas.

#### 2.4.5.3 Problema en la implementación utilizando JavaScript

Durante la implementación del proyecto utilizando lenguaje JavaScript se desarrollaron actividades que presentaron inconvenientes en el desarrollo del proyecto, problemas los cuales se encontraron alternativas o soluciones que permitieron el desarrollo de una implementación exitosa con este lenguaje, algunas de las situaciones fueron las siguientes:

Establecimiento de conexión con el servicio en la ruta para enseñar el teclado virtual. La solución fue realizar una solicitud *fetch* para realizar la solicitud HTTP GET a la URL donde se encuentra el teclado y posteriormente ser publicado en un *div* donde este será almacenado y publicado.

|  |
| --- |
| function cargarContenido() {  // Realizar una solicitud HTTP GET a la URL que alberga el teclado  fetch('http://localhost:8080/keyboard')  .then(response => response.text())  .then(data => {  document.getElementById('keyboard-container').innerHTML = data;  connect();  })  .catch(error => console.error('Error:', error));  } |

Establecer una conexión con el Websocket, desarrollado en Spring Boot para permitir una conexión en tiempo real con el servidor. Como solución a este inconveniente se desarrolló una función para la conexión que sería ejecutada al momento de cargar el teclado virtual, esta función es llamada *connect()* que cumple la función de establecer una conexión con la base de datos para poder manejar los websocket.

|  |
| --- |
| *var* stompClient = null;  *function* connect() {  // Parte #1  console.log(typeof SockJS);  *//URL para conección al endpoint WebSocket*  *var* socket = new SockJS('http://localhost:8080/ws-chat');  stompClient = Stomp.over(socket);  stompClient.connect({}, *function* (*frame*) {  console.log('Conectado: ' + frame);  });  // Parte #2  *// Suscribirse al canal después de la conexión exitosa*  stompClient.subscribe('/topic/message', *function* (*response*) {  *var* message = JSON.parse(response.body);  *// Manejar el mensaje recibido y mostrarlo en el chat-container*  *var* chatContainer = document.getElementById('chat-container');  chatContainer.innerHTML += '<p>' + message.message + '</p>';  });  } |

Dependencias necesarias para utilizar SockJS y StompJS:

|  |
| --- |
| <script *src*="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.6.1/sockjs.min.js" *integrity*="sha512-1QvjE7BtotQjkq8PxLeF6P46gEpBRXuskzIVgjFpekzFVF4yjRgrQvTG1MTOJ3yQgvTteKAcO7DSZI92+u/yZw==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script>      <script *src*="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.6.1/sockjs.js" *integrity*="sha512-aHuAnb+FZ/yM5zYvdiA8q0sxLdsBQ3PRwO8D/08qsenCLQ2gWsosB43Cn1X8esAz0IpaC7Ns5cAPyKIdJQ0hNw==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script>      <script *src*="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js" *integrity*="sha512-iKDtgDyTHjAitUDdLljGhenhPwrbBfqTKWO1mkhSFH3A7blITC9MhYon6SjnMhp4o0rADGw9yAC6EW4t5a4K3g==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script>      <script *src*="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.js" *integrity*="sha512-tL4PIUsPy+Rks1go4kQG8M8/ItpRMvKnbBjQm4d2DQnFwgcBYRRN00QdyQnWSCwNMsoY/MfJY8nHp2CzlNdtZA==" *crossorigin*="anonymous" *referrerpolicy*="no-referrer"></script> |

Esta función es utilizada para establecer comunicación en tiempo real utilizando la variable *stompClient* usando StompJS. Para manejar la conexión, en la segunda parte del código se realiza una conexión a la dirección */topic/message* del servidor creado al ejecutar Spring Boot, para verificar una conexión exitosa se implementa un *console.log* que publica un mensaje con el estado de la solicitud de la conexión.

Realizar una distinción entre los inputs donde se publicarán los valores del teclado. Se implementan una serie de funciones para diferenciar comenzando con una función para albergar la id del input elegido.

|  |
| --- |
| *//Los inputs deben implementar la función SelectInput enviando el valor de identidad del mismo*  *onclick*="SelecionInput(*this*)" |

En la función se alberga el valor del botón dentro de un variable que tendrá la identidad del input elegido además se verifica que cumpla la condición de haber seleccionado la opción CheckBox que solicita y enseña el teclado virtual.

|  |
| --- |
| *var* inputIDs = []; *// Array para almacenar las ID de los inputs*  *function* SelecionInput(*inputElement*){  *// Obtén el valor del id del input*  *var* idInput = inputElement.id;                  inputIDs.splice(0, inputIDs.length);                  inputIDs.push(idInput);                  console.log("input:: "+ inputIDs);  *// Obtén el estado del checkbox*  *var* checkbox = document.getElementById("discapacidad");  *var* divKeyB = document.getElementById("keyboard-container");                  if (checkbox.checked) {  *// Si el checkbox está seleccionado, muestra el teclado*                      console.log('check');                      divKeyB.style.visibility ="visible";                  } else {  *// Si el checkbox no está seleccionado, oculta el teclado*                      console.log('not check');                      divKeyB.style.visibility ="hidden";                  }        } |

Para diferenciar entre los inputs se utiliza la variable *inputIDs* que almacena la identidad del input seleccionado (valor que se actualiza conforme se elige otro input) y durante el uso de la función *LscKeyPressed* (ejecutada por los botones del teclado) una nueva variable toma el valor del botón seleccionado en el arreglo que será utilizado para identificar el destino de las elecciones de letras echas con el teclado virtual.

|  |
| --- |
| *function* LscKeyPressed(*button*) {  *var* dataKey = button.getAttribute('data-key');  *var* messageInput = document.getElementById(inputIDs);              console.log("input:: "+ inputIDs);  *// Verificar si dataKey es "del" y eliminar el último carácter*              if (dataKey === "del") {  *var* inputValue = messageInput.value;                  if (inputValue.length > 0) {  *// Eliminar el último carácter*                      messageInput.value = inputValue.slice(0, -1);                  }              } else {  *// Agregar el valor de dataKey al input-field*                  messageInput.value += dataKey;              }              } |

#### 2.4.5.4 Problema en la implementación utilizando PHP

Los problemas que se han descubierto durante la implementación del servicio Spring Boot utilizando el lenguaje PHP es el hecho de la necesidad del uso de un servicio web que permita la ejecución de este lenguaje. Mediante diferentes investigaciones se encontró una solución que permitía el uso de características del lenguaje HTML para realizar la solicitud y publicación del teclado virtual además de que este es funcional y permite la interacción y publicación por consola de los valores de los botones seleccionados.

Solicitud usando HTML.

|  |
| --- |
| <!-- Solicitud HTML del teclado virtual--!> <iframe src="http://localhost:8080/keyboard" frameborder="0"></iframe> |

Solicitud usando PHP.

|  |
| --- |
| <div class="input-field" id="keyboard-container">  <?php  //Obtener el contenido de "<http://localhost:8080/keyboard>" utilizando PHP  $contenido = file\_get\_contents('http://localhost:8080/keyboard');  // Enseñar el contenido dentro de la página  echo $contenido;  ?>  </div> |

A su vez también debido a las características y finalidades del lenguaje PHP que es un lenguaje orientado a la comunicación con el servidor este no posee las características que permitan crear funciones para la manipulación de los objetos de HTML. Debido a eso, se establece la necesidad del uso de un lenguaje orientado al lado del cliente para la creación de funciones necesarias para realizar actividades. Pese a poder implementarse este lenguaje, debe recordarse los requerimientos ya mencionados el uso de web service para la ejecución del lenguaje PHP y que debe de combinarse con otros lenguajes que permitan la ejecución de funciones para poder implementar otras aplicaciones definidas en Spring Boot.

## 2.5 Pruebas

### 2.5.1 Comprobación de tecnologías necesarias.

Las tecnologías necesarias para que nuestro microservicio funcione son el websocket y el Swagger UI.

El WebSocket es utilizado en nuestro proyecto para establecer una conexión con la base de datos para lo cual se implementan las dependencias de StompJS y SockJS. Las dependencias mencionadas se destinan a ayudar a la conexión e implementación de los websockets siendo StompJS para él envió de mensajería al servidor y SockJS para la implementación del websocket. Para verificar la conectividad con SockJS se usa la dirección *ws-chat* y StompJS se implementa en las páginas web las dependencias necesarias para poder implementar la función StompJS.

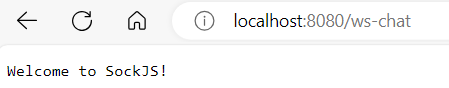


Ilustración 14. Conexión a endpoint usando SockJS

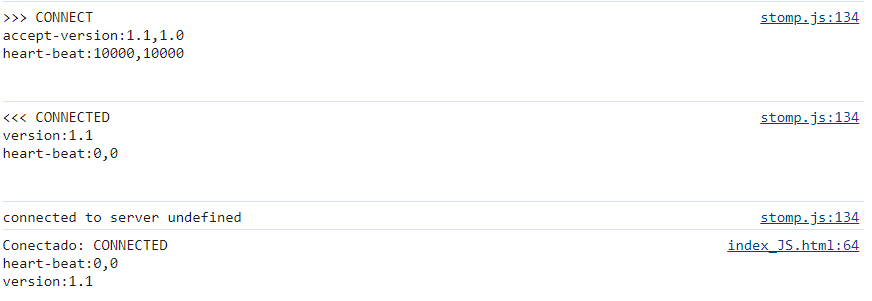


Ilustración 15. Conexión websocket implementando StompJS

La herramienta Swagger funciona como un entorno grafico para la implementación de funcionalidades del API desde Spring Boot. usando este servicio se probaron las diferentes funciones desarrolladas por la facilidad que brinda el entorno gráfico. Como se aprecia en las siguientes imágenes esta interfaz se adapta a las funciones haciendo referencia a las direcciones definidas.

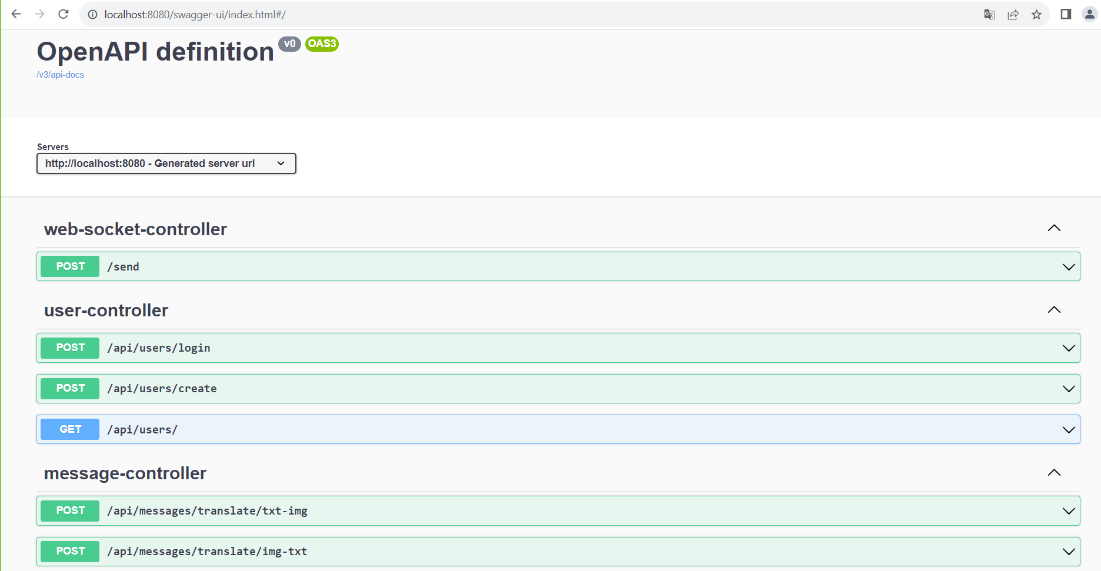


Ilustración 16. Swagger funcionalidades 1

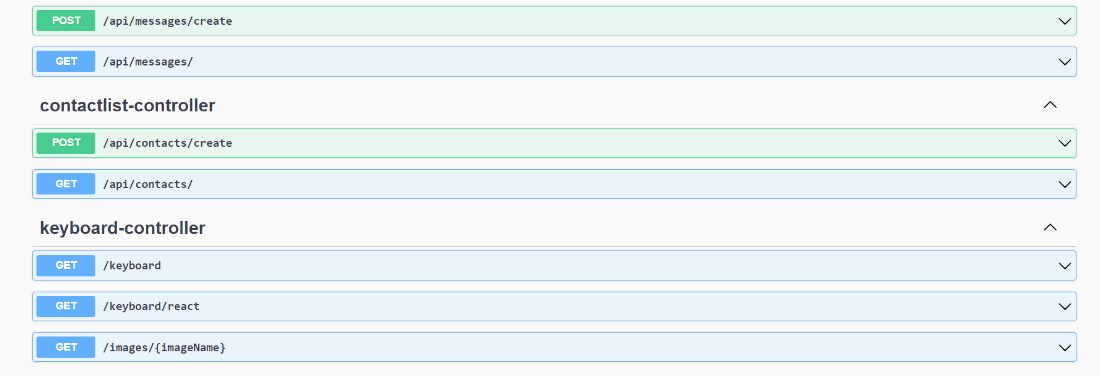


Ilustración 17. Swagger funcionalidades 2

Estas imágenes son la interfaz de Swagger donde se enseñan las peticiones diseñadas en el API. Estas funciones pueden ser llenadas con la información necesaria para realizar las pruebas que quiera hacer quien esté utilizando el proyecto

### 2.5.2 Comprobación del comportamiento de la base de datos.

La base de datos implementada en el proyecto se instaló al generar el documento en forma de dependencia siendo esta H2. Esta es la base de datos embebida en la aplicación, la cual es utilizada para realzar pruebas sobre el modelo creado, y las funciones definidas.

Ilustración 18. Configuración proyecto dependencia H2

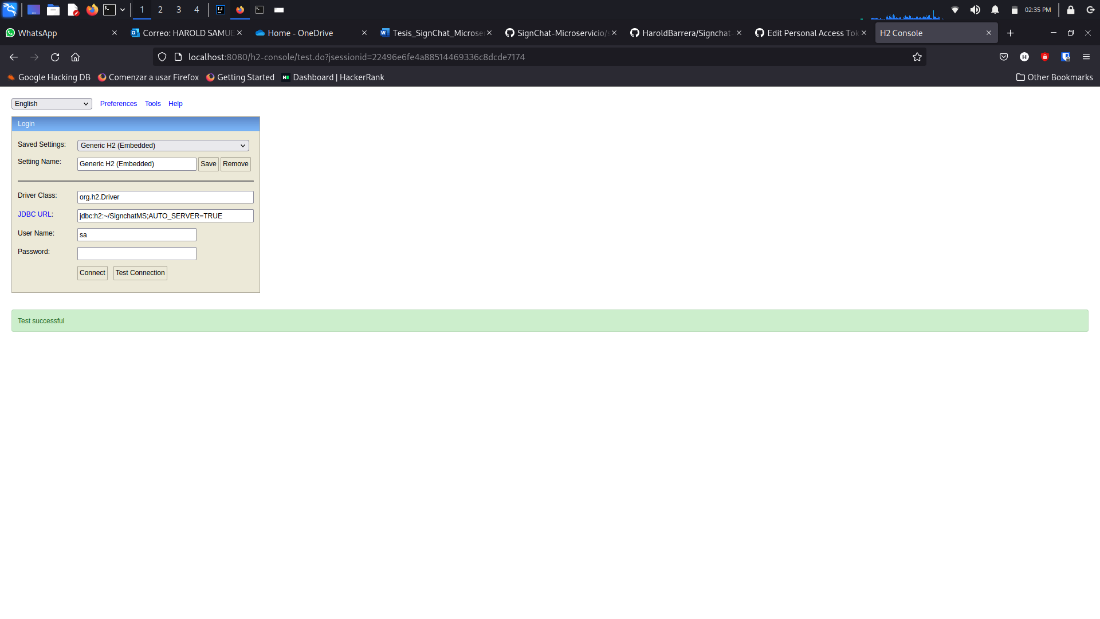


Ilustración 19. Comprobación conexión de base de datos

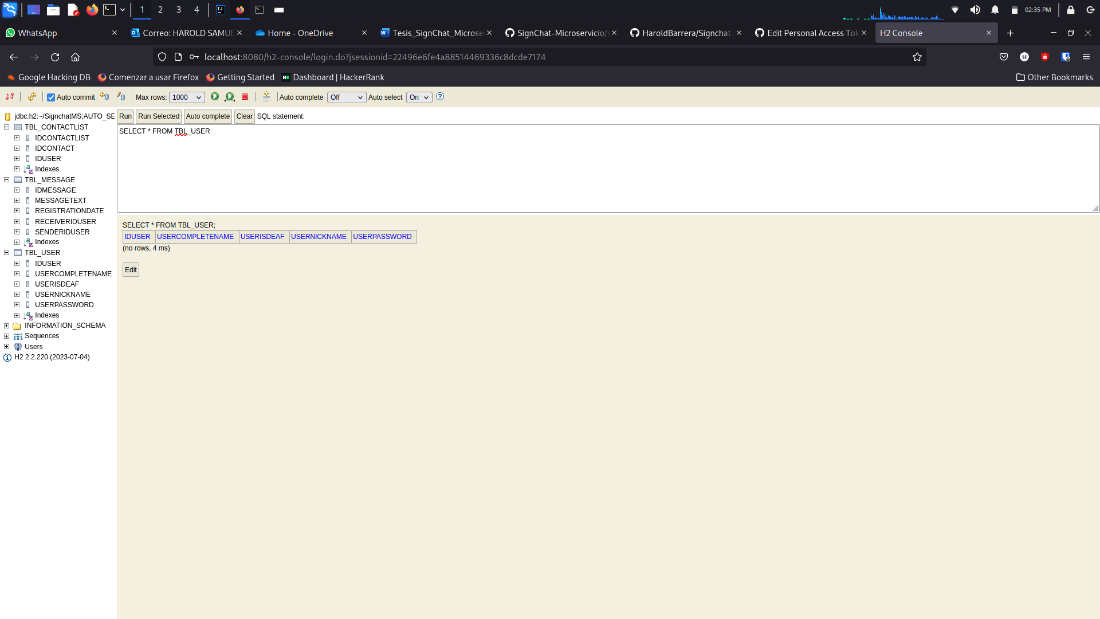


Ilustración 20. Funcionamiento de Base de datos

En esta última imagen de la base de datos temporal que se ha creado donde se lista la estructura y se visualiza la interfaz del editor de la base de datos. Se puede ver la forma en cómo se publican las sentencias de consulta de información y la forma en que se listan los resultados, con esto se observa la forma que se almacena y organiza la información ingresada.

### 2.5.3 Comprobación de funcionalidad.

Se mostrarán fotos del funcionamiento del proyecto y se profundarán más en el título de páginas HTML de prueba. En la siguiente imagen se muestra el funcionamiento del teclado LSC.

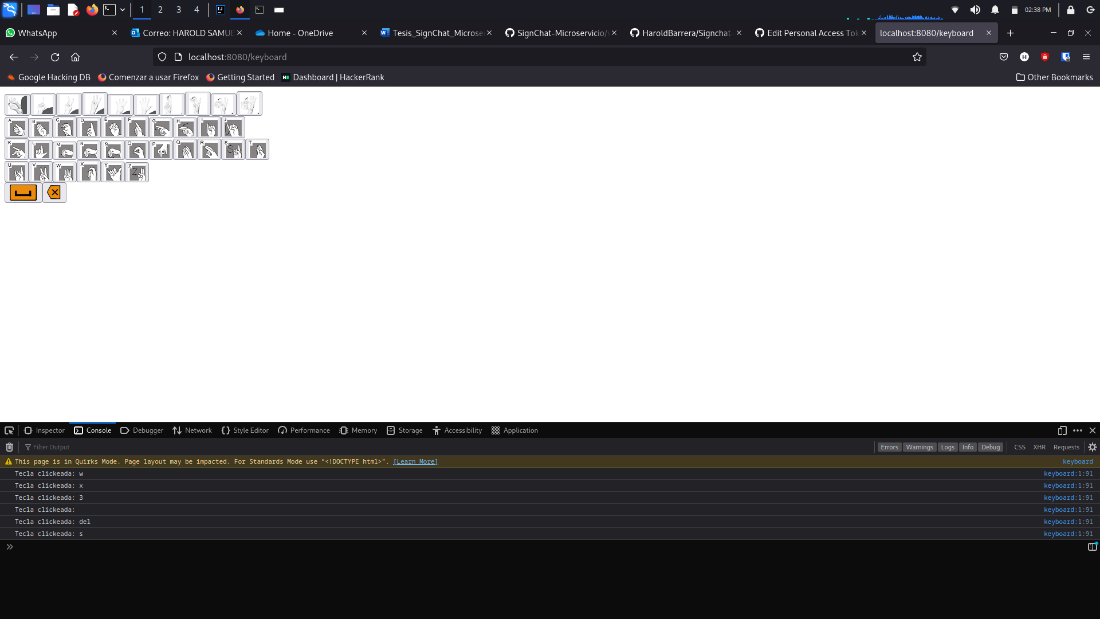


Ilustración 21. Visualización de teclado virtual.

## 2.6 Despliegue

### 2.6.1 Creación del plugin.

Para la creación del plugin no se requiere de alguna tecnología aparte o configuración externa, puesto que se puede hacer desde el propio IDE. En nuestro caso, IntelliJ permite crear el plugin de manera sencilla con el menú “*Build*” que ofrecen.

|  |  |
| --- | --- |
| **Nombre del producto** | SignChatMS.jar |
| **Funcionalidades** | * Teclado LSC. * Traducción de mensajes de imagen LSC a texto y viceversa. * Mensajería en tiempo real. |

Tabla 2. Resultado del producto.



Ilustración 22. Plugin de SignChat.

### 2.6.2 Lanzamiento del microservicio.

El microservicio será lanzado de una página de Hosting Web gratuito llamador render.

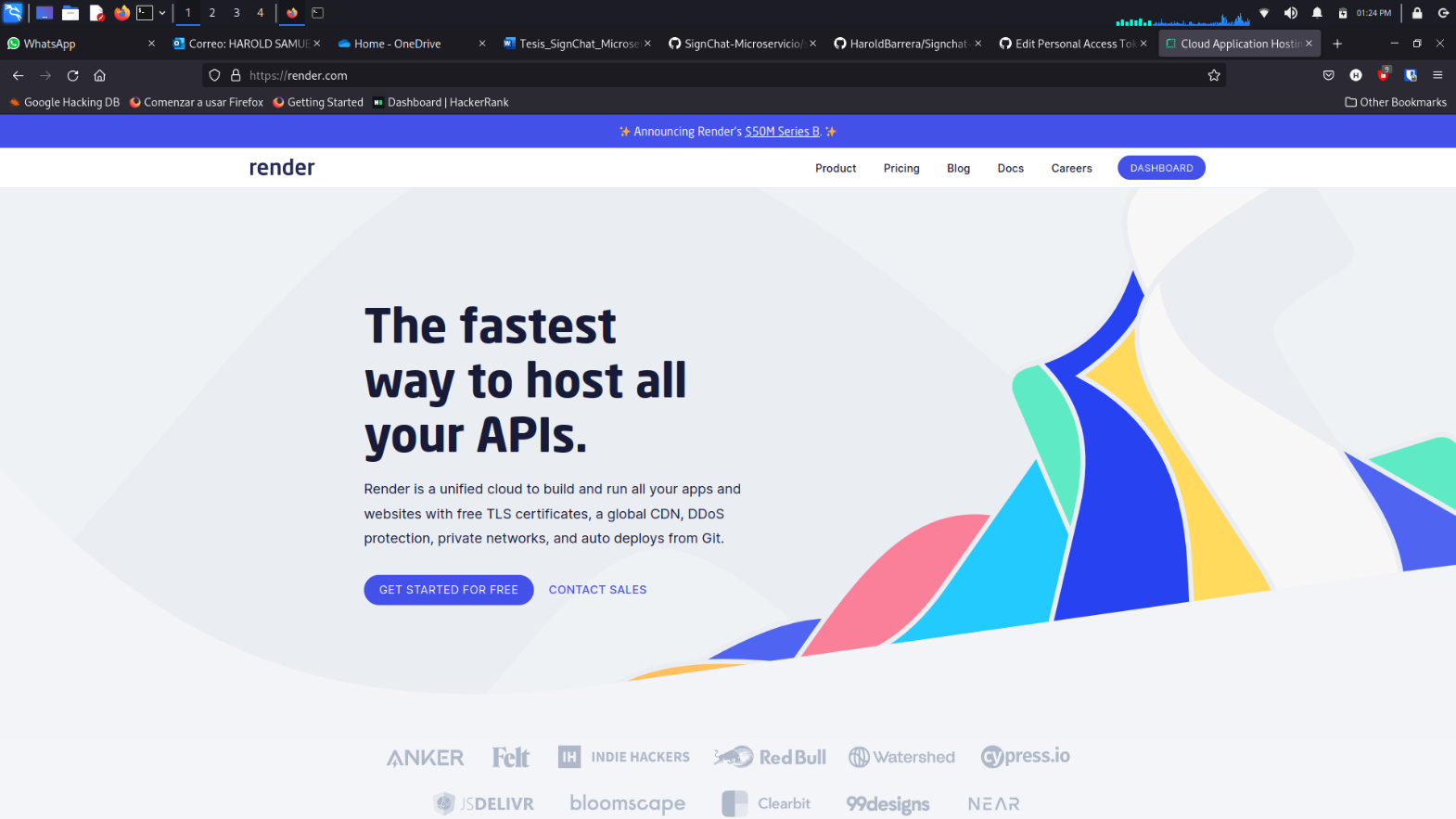


Ilustración 23. Render.

Con ello, podremos convertir nuestro microservicio en un web service para que todos puedan usarlo mediante una URL. El proceso es bastante sencillo, ya que render solo necesita el repositorio de GitHub y una serie de instrucciones para iniciar el servicio. Para esto último utilizaremos Docker, el cual es una tecnología que nos permitirá, en el caso de nuestro proyecto, generar la serie de instrucciones para que Render publique nuestra aplicación a la web.

Creamos nuestro archivo Dockerfile, especificándole la versión de Java, en que sistema operativo correrá la aplicación y entre oros detalles necesarios para ejecutar el programa.

#Entorno de trabajo

FROM maven:3.8.3-openjdk-17 AS build

#Copiar todos los archivos del proyecto

COPY . .

#Paquetes de maven

RUN mvn clean package -DskipTests

# Etapa de ejecución

FROM openjdk:17-jdk-slim

# Copiar el archivo JAR construido desde la etapa de construcción

COPY --from=build /target/SignchatMS-0.0.1-SNAPSHOT.jar SignchatMS.jar

# Exponer el puerto 8080 para que la aplicación sea accesible

EXPOSE 8080

# Comando para ejecutar la aplicación Spring Boot

ENTRYPOINT ["java", "-jar", "SignchatMS.jar"]

Una vez creado nuestro Dockerfile, nos dirigimos a render y creamos un nuevo web service usando el repositorio de GitHub del microservicio.



Ilustración 24. Configuraciones hechas en Render.

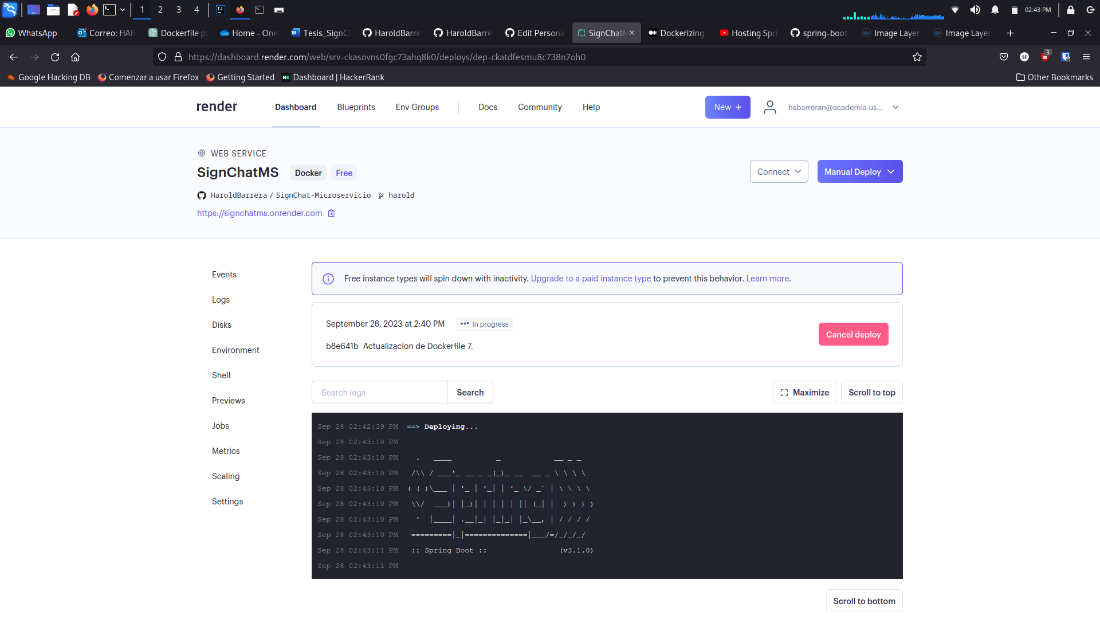


Ilustración 25. Microservicio lanzado en Render.

Ahora el microservicio se encuentra en la web y cualquiera puede acceder a las rutas de este.

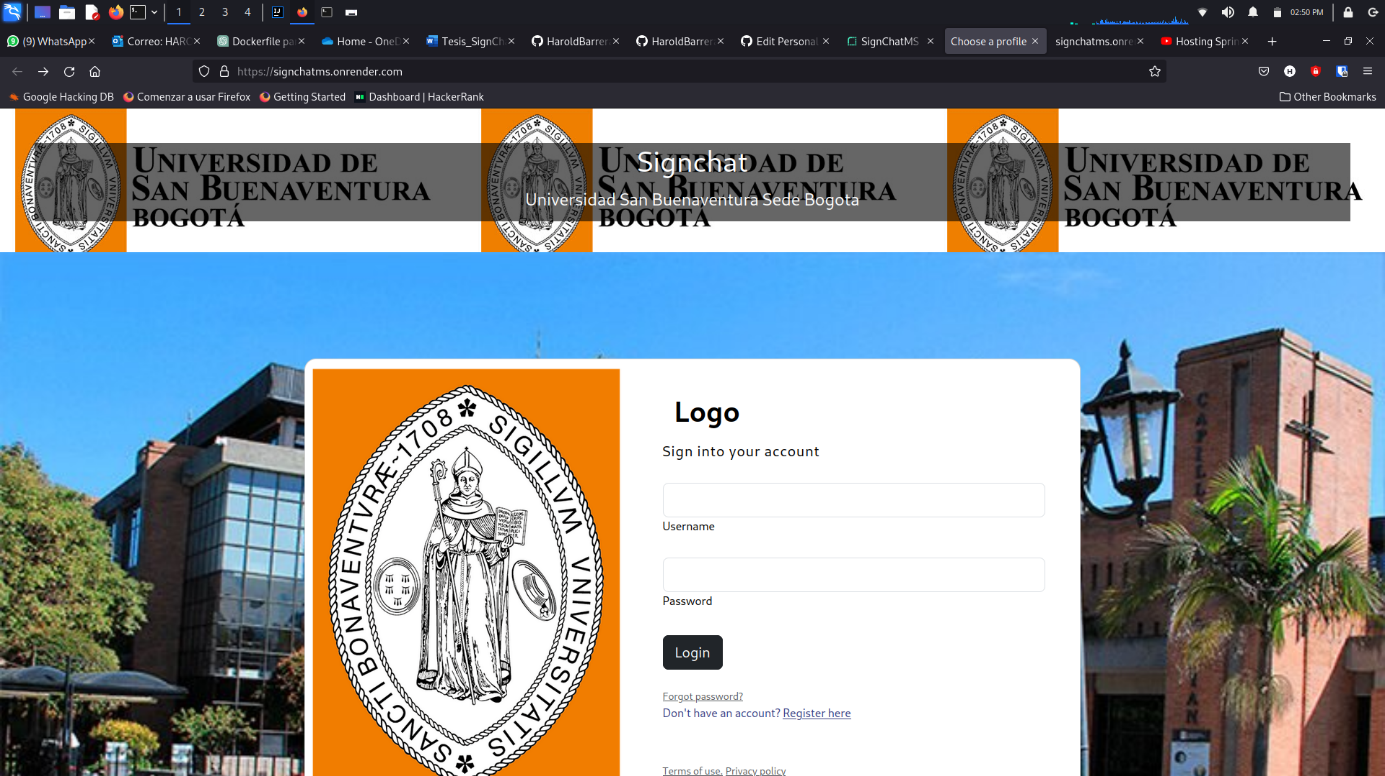
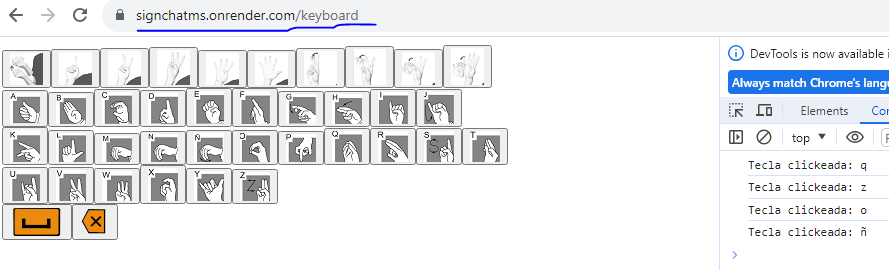


Ilustración 26. Microservicio con Host en Render.



# Capítulo 3. Análisis de Resultados.

En esta sección mostraremos los resultados finales del microservicio de SignChat junto a una página web de prueba con la finalidad de mostrar de forma gráfica el funcionamiento de las funcionalidades y la forma de interacción en un entorno de prueba donde se simularán situaciones donde serían utilizados.

## 3.1 Microservicio de SignChat.

Procederemos a mostrar los resultados del proyecto final, a continuación, la pantalla principal.

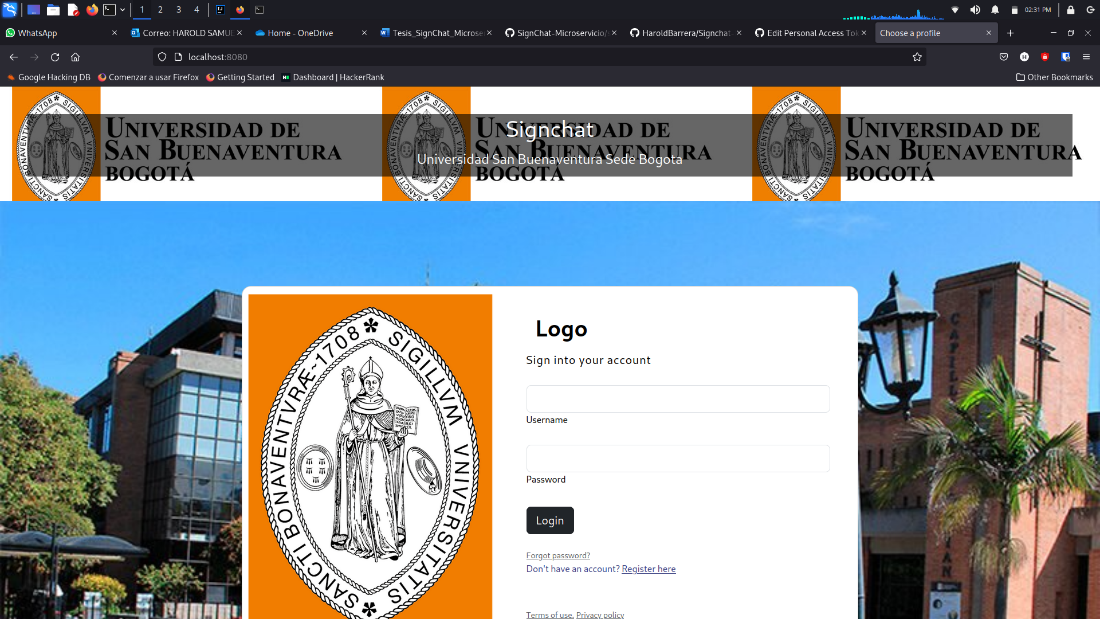


Ilustración 27. Pantalla principal.

Esta pantalla no tiene funcionalidades en sí, solo sirve para mostrar una plantilla de página de inicio para alguna aplicación.

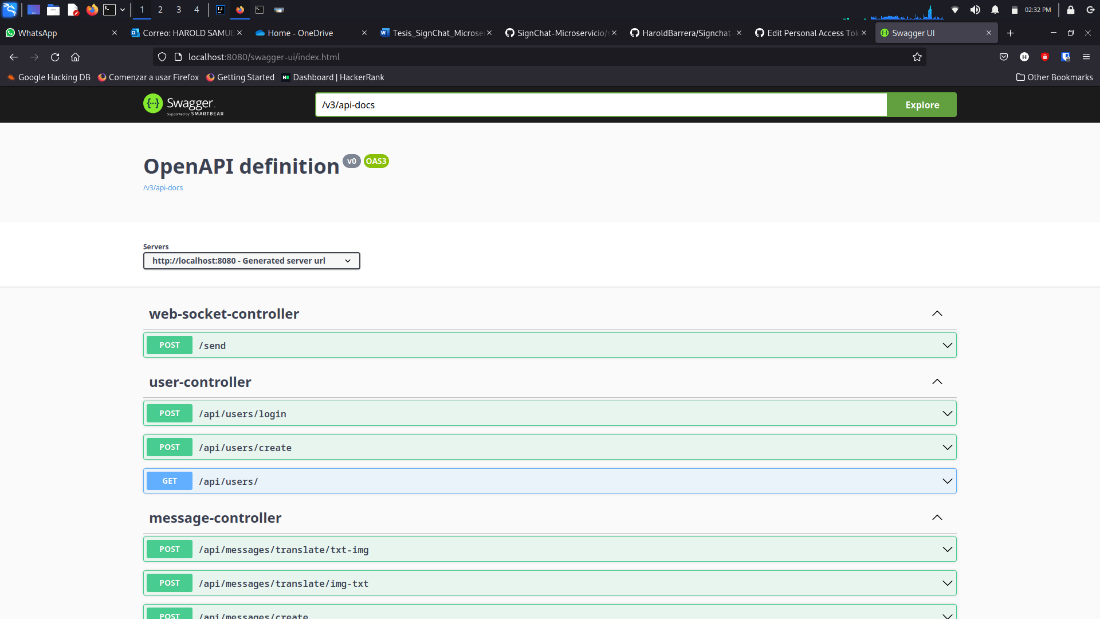


Ilustración 28. Swagger UI.

Esta pantalla es la ruta para Swagger, donde nos muestra todas las funcionalidades del microservicio con la posibilidad de probarlos mediante el formato JSON. En esta pantalla también podemos observar los esquemas, que son los modelos de nuestra base de datos.

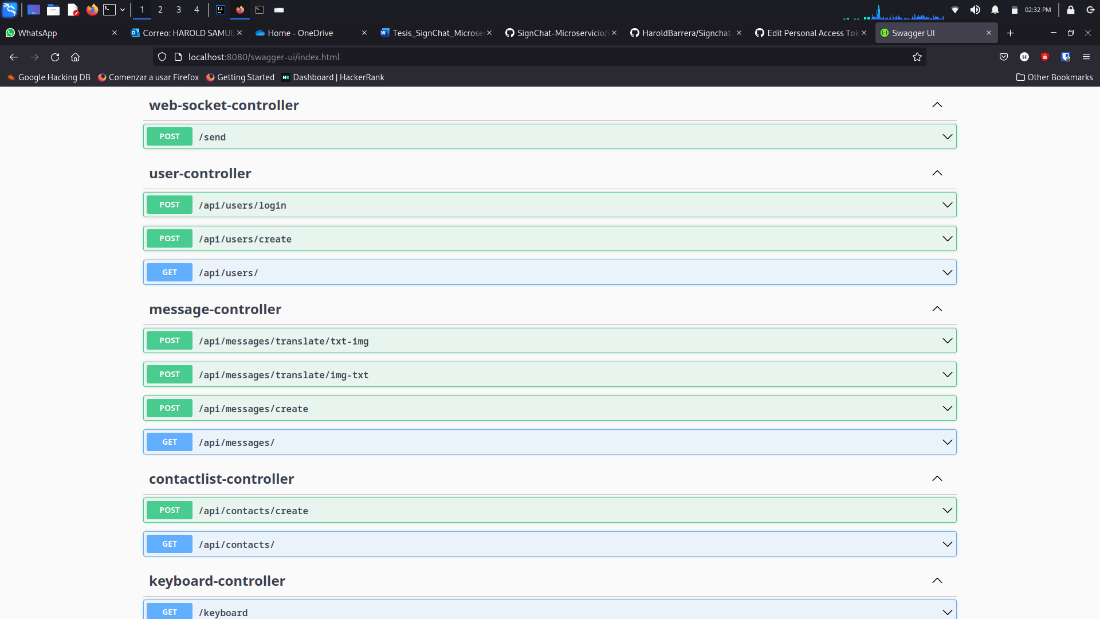


Ilustración 29. Swagger UI, funcionalidades.

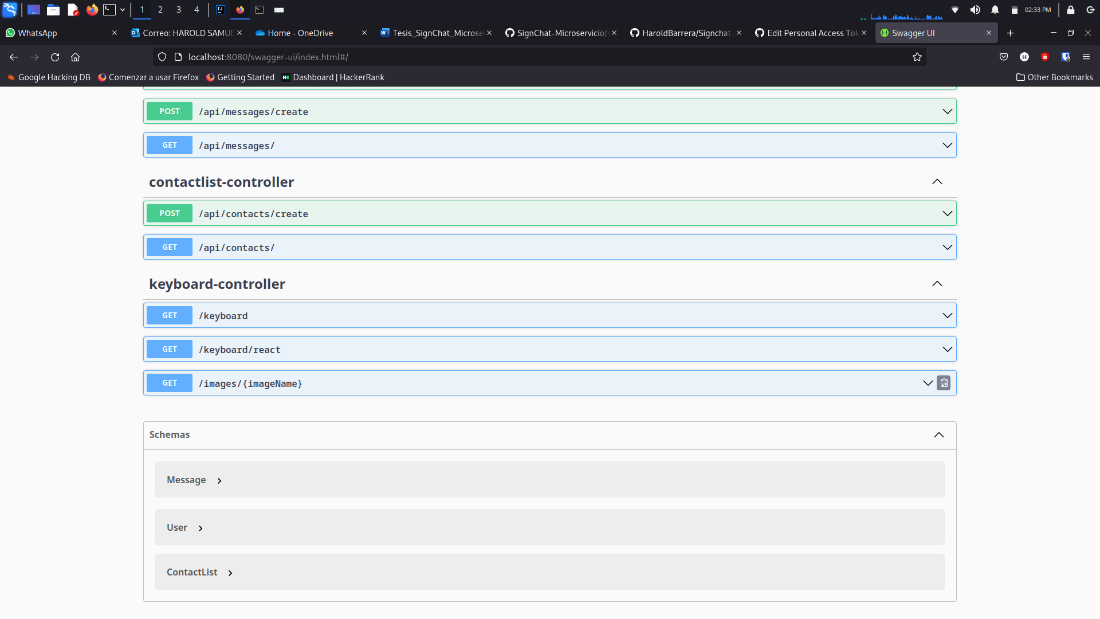


Ilustración 30. Swagger UI, Esquemas.

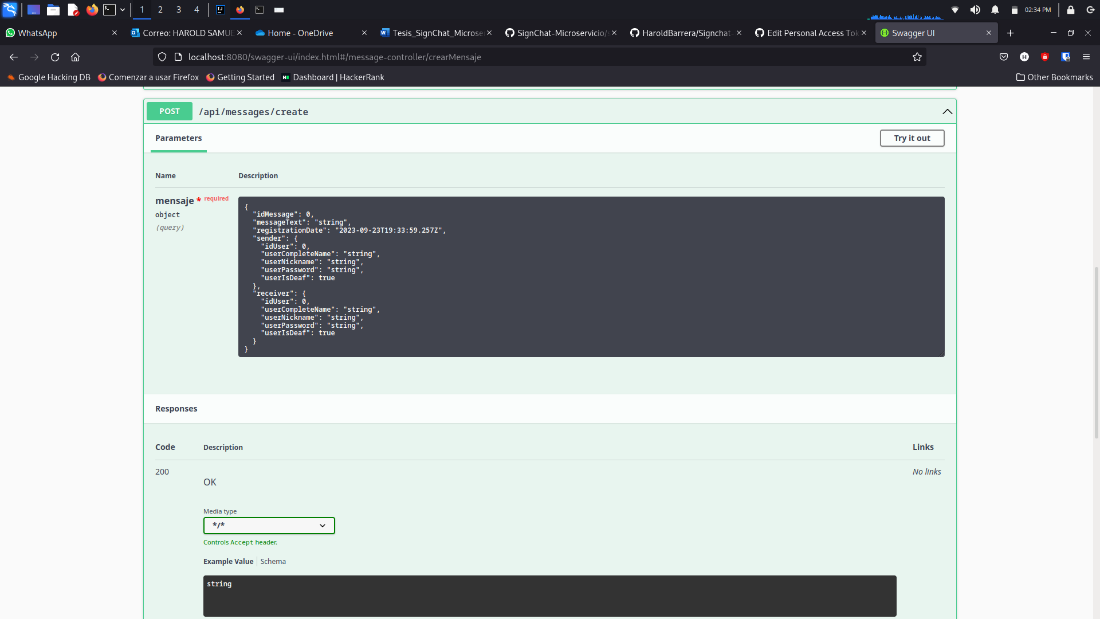


Ilustración 31. Swagger UI, prueba de API REST.

Ahora nos dirigimos a la pantalla principal de H2. En esta ruta observamos que primero nos pedirá un usuario y contraseña (los cuales especificamos en la configuración del proyecto). Después de comprobar que el usuario y contraseña es correcto, podremos observar nuestra base de datos, con sus respectivas tablas y datos (aunque por el momento no se ha creado ningún dato).

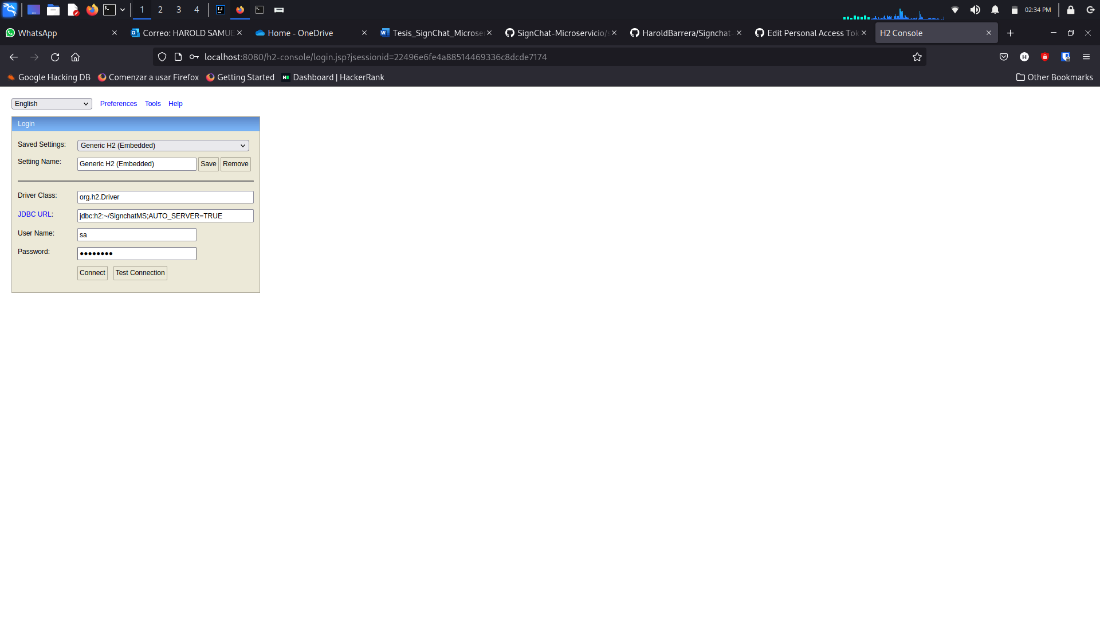


Ilustración 32. Inicio de sesión de base de datos.

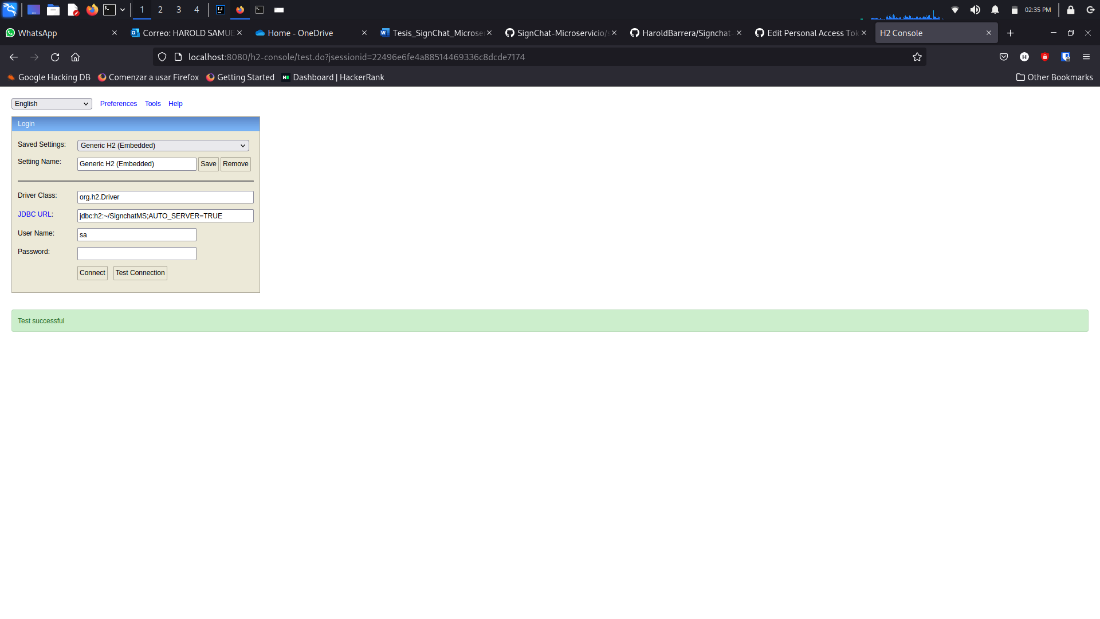


Ilustración 33. Conexión exitosa de base de datos.

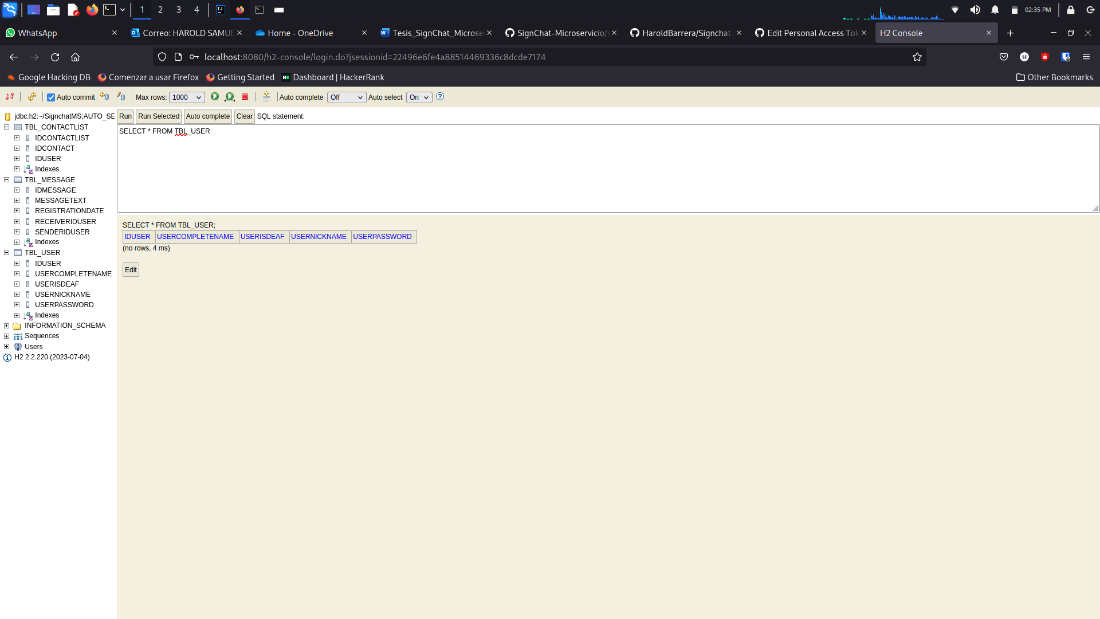


Ilustración 34. Interfaz de Base de datos.

Ahora vamos a visitar las dos rutas más importantes del proyecto. La primera de ellas es la ruta donde se inicia el websocket. En esta ruta no hay mucho además de un mensaje para confirmar que el websocket este encendido.

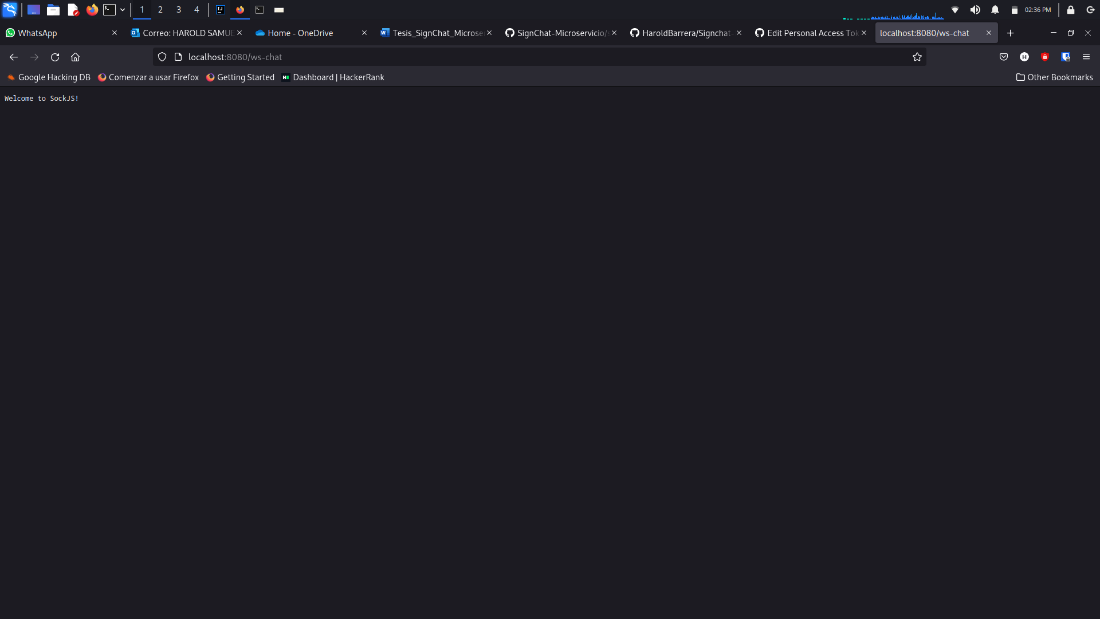


Ilustración 35. Websocket.

Ahora tenemos la segunda ruta que es donde se llama o se invoca el teclado de lenguaje de señas colombiano. En esta ruta podemos apreciar no solo el teclado, sino que también si entramos a la consola del navegador (F12), podemos ver que nos informa que tecla ha sido pulsada.

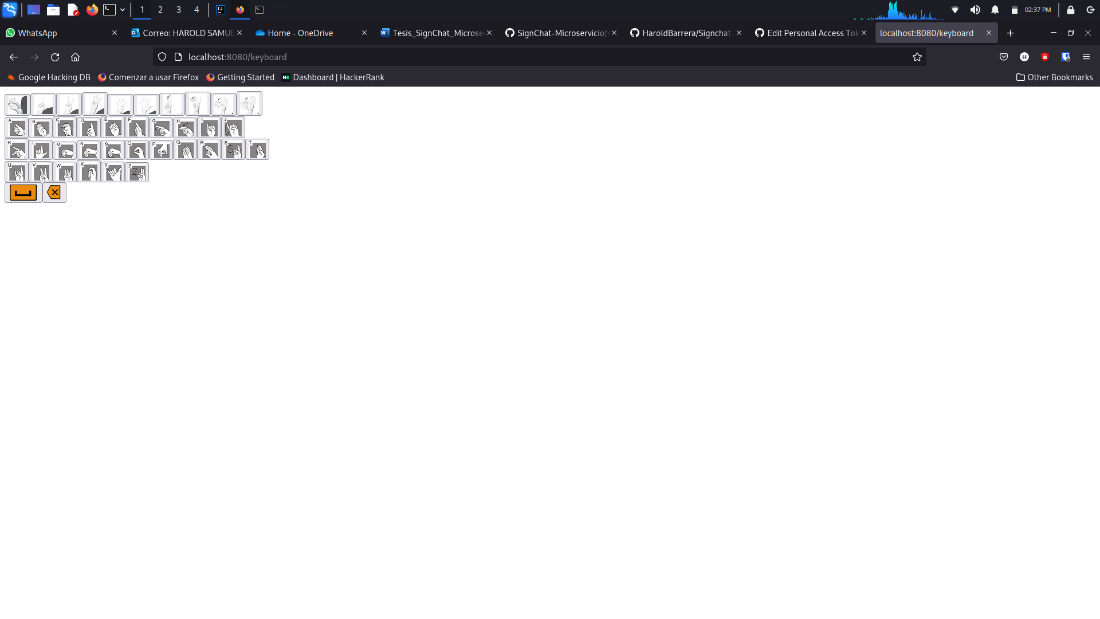


Ilustración 36. Teclado virtual LSC.

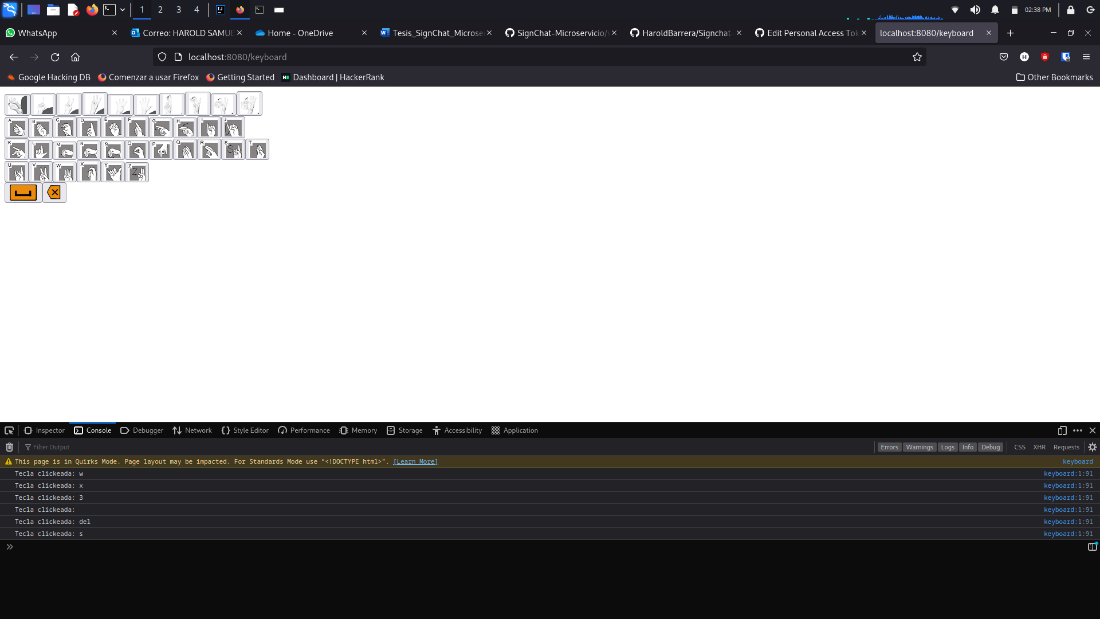


Ilustración 37. Teclado LSC en acción.

## 3.2 Página Web de prueba.

Para la prueba de funcionamiento de las actividades y funciones que se realizan se ha desarrollado un modelo de prueba de una página HTML que implementa las actividades propuestas, se ha utilizado el lenguaje JavaScript para la presentación de las actividades.

Al cargar el teclado virtual se ejecuta una solicitud de conexión al servidor temporal creado por Spring Boot.

|  |
| --- |
| Ilustración 38. Mensaje de conexión con base de datos temporal |

La primera actividad es el publicar el teclado virtual que se ha desarrollado además que este debe de ser funcional lo que implica que todos sus componentes deben cumplir una función además que se debe de publicar las etiquetas(imágenes) que hagan referencia del valor asignado a cada botón.

|  |
| --- |
| Ilustración 39. Teclado de señas virtual, con espacio de texto vacío |

Como se evidencia en la imagen anterior el teclado se encuentra formado por la escala numérica de 0-9 seguido del alfabeto y por último botones que simbolizan las acciones de espacio y eliminar, cada uno con una imagen que permite la identificación clara de su valor, en la siguiente ilustración se evidencia como se ve un texto publicado en el cuadro de texto.

|  |
| --- |
| Ilustración 40. Teclado de señas virtual, con espacio de texto lleno |

A su vez existe un botón para publicación del mensaje definido en el cuadro de texto, la publicación del mensaje es un cuadro inalterable, además que se puede enviar varios mensajes y no se eliminan.

|  |
| --- |
| Ilustración 41. Pagina HTML de prueba que implementa teclado y su funcionamiento |

También se han creado modificaciones donde en caso de la existencia de más de un cuadro de texto se realiza una diferenciación sobre en qué cuadro será publicado las elecciones del teclado.

|  |
| --- |
| Ilustración 42. Pagina HTML implementa más input |

## CONCLUSIONES

En conclusión, los resultados que se tenían previstos en el proyecto como la capacidad de poder implementar el servicio en diferentes páginas web, desarrollo de teclado virtual que implementa lenguaje de señas colombiano y la traducción de los textos de imagen a letras y viceversa han sido desarrollados de manera exitosa. Se logró la creación de una API, la cual se desarrolló implementando tecnologías que facilitaron su desarrollo, configuración e implementación. Se desarrolló una documentación sobre los métodos de implementación del entregable con diferentes leguajes y se adjuntaron pruebas de sus funcionamientos en entornos que implementan el mismo y utilizan las funcionalidades y requerimientos especificados en este documento.

Las pruebas de funcionamiento que se realizaron para establecer evidencia del funcionamiento de las actividades. esto es usado como evidencia de resultados. Durante este desarrollo se realizaron investigaciones sobre las mejores herramientas que pueden ser utilizadas o ser útiles en el desarrollo, pero también se realizaron cambios en herramientas que en un inicio se pensó que serían útiles, pero luego generaron conflicto en su implementación.

Esta fue una configuración estándar desarrollada, pero con la flexibilidad que se ofrece es posible agregar o modificar dependencias.

Con desarrollo de las pruebas técnicas demostramos que la API desarrollada puede ser implementada en diferentes modelos de páginas web o lenguajes de desarrollo. En estos fragmentos evidenciamos la capacidad adaptativa para la implementación en diferentes entornos y disponibilidad de los servicios y funciones ofrecidas como proporcionar al proyecto la traducción de texto, creación de una base de datos, interfaces y modelos CRUD para el desarrollo de funciones como crear, actualizar, borrar y visualizar los datos de la base de datos de la aplicación. También, otra funcionalidad que se desarrollo es la configuración de un CORS para que cualquier aplicación web pueda acceder a las funcionalidades en caso de que la pagina que solicita la conexión en caso de que el API no este es su dominio.

Se ha de señalar que para las personas que quieran hacer futuras implementaciones, las posibilidades de mejora y personalización del proyecto, según las necesidades de los usuarios, existen varias opciones como la posibilidad de personalizar el modelo de las imágenes del teclado por un conjunto que se encuentre más orientado a lo buscado, o la capacidad de implementar el teclado y su capacidad de traducción en páginas ya desarrolladas con la finalidad de mejorar la atención a usuarios y ampliar el público al que se puede llegar teniendo estas características de accesibilidad.

# REFERENCIAS

Blankenberg, D., Kuster, G. V., Coraor, N., Ananda, G., Lazarus, R., Mangan, M., ... & Taylor, J. (2010). Galaxy: a web‐based genome analysis tool for experimentalists. Current protocols in molecular biology, 19-10.

Bolger, A., & Giorgi, F. Trimmomatic: A Flexible Read Trimming Tool for Illumina NGS Data. URL http://www. usadellab. org/cms/index. Php.

HTTP - Concepto, para qué sirve y cómo funciona. (n.d.). Concepto. <https://concepto.de/http/#ixzz7wh3wKjK9>

¿Qué es una API de REST? (n.d.). <https://www.redhat.com/es/topics/api/what-is-a-rest-api#resumen>

Métodos HTTP. (n.d.). <https://diego.com.es/metodos-http>

Ciclo de vida del desarrollo de software (SDLC) explicado -Ungoti. (2021, October 24). Ungoti. <https://ungoti.com/es/soluciones/desarrollo-de-software/sdlc/#:~:text=El%20ciclo%20de%20vida%20del,definici%C3%B3n%20de%20los%20requisitos%20hasta>

Aula. (2023). ¿Qué es Java y para qué sirve? *aula21 | Formación para la Industria*. <https://www.cursosaula21.com/que-es-java/>

*¿Qué son los microservicios?* (s. f.). TIBCO Software. <https://www.tibco.com/es/reference-center/what-are-microservices>

Gustavo. (2021). Que es Spring Boot. *Home*. <https://gustavopeiretti.com/spring-boot-que-es/>

Gonzalez, M. V. (2021). ¿Qué es Spring Boot? *Codmind*. <https://blog.codmind.com/que-es-spring-boot/>

Valverde, D. (2017). ¿Qué es Spring? *David Valverde Blog*. <https://www.davidvalverde.com/blog/que-es-spring/>

Baeldung, & Baeldung. (2023). Setting up Swagger 2 with a spring REST API | Baeldung. *Baeldung*. <https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api>

*H2 Database Engine*. (s. f.). <https://www.h2database.com/html/main.html>

*Firebase Realtime database*. (s. f.). Firebase. <https://firebase.google.com/docs/database?hl=es-419>

*WebSockets - referencia de la API Web | MDN*. (2023, 21 febrero). <https://developer.mozilla.org/es/docs/Web/API/WebSockets_API>

*¿Cómo puedo instalar Java?* (n.d.). <https://www.java.com/es/download/help/download_options.html>

*Git - Instalación de Git*. (n.d.). <https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Instalaci%C3%B3n-de-Git>

Atlassian. (n.d.). *Instalación de Git | Atlassian Git Tutorial*. <https://www.atlassian.com/es/git/tutorials/install-git#windows>

Atlassian. (n.d.). Qué es Git | Atlassian Git Tutorial. https://www.atlassian.com/es/git/tutorials/what-is-git

*¿Cómo puedo descargar e instalar Java en un equipo con Windows de forma manual?* (n.d.). <https://www.java.com/es/download/help/windows_manual_download.html>

*¿Cómo puedo instalar Java en mi Mac?* (n.d.). <https://www.java.com/es/download/help/mac_install.html>

**ANEXOS**

En esta sección se mostrará la instalación de las diferentes herramientas utilizadas para la creación del microservicio.

# Anexo 1. Instalación de IntelliJ

Para el desarrollo del proyecto se utiliza IntelliJ IDE diseñado para el lenguaje Java y a demostrado ser muy eficiente a lo largo de los años. Existen dos versiones “Ultimate” y “Community edition” siendo esta ultima la versión gratuita y la que se decidió utilizar para el desarrollo del proyecto, para la instalación del programa se ingresa en la página oficial y se descarga la versión Community Edition.

#### Anexo 2. Instalación de GIT

El proceso de instalación de GIT difiere según el sistema operativo, en caso del sistema operativo Linux se debe usar los siguientes comandos:

|  |
| --- |
| Comando para la instalación en Linux   yum install git  En caso de usar una distribución Ubuntu:  apt-get install git |

En MAC se puede hacerlo desde la página de GIT, en caso de Windows se realiza al visitar la página oficial de Git para descargar la versión que se adapte a la máquina, también se puede descargar mediante la terminal de comandos.

|  |
| --- |
| winget install --id Git.Git -e --source winget |

**Anexo 3. Instalación de lenguaje de JAVA 17**

La instalación del lenguaje JAVA varía dependiendo del sistema operativo que se utiliza

* Windows: se debe descargar el instalador en la página oficial y al ejecutarlo se debe de seguir las indicaciones para la configuración.

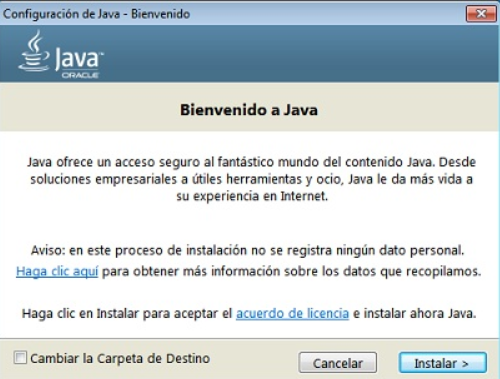


Ilustración 43. Instalador Java para Windows.

* Mac: par instalación y configuración del lenguaje en este sistema operativo se debe descargar el archivo jre-8u311-macosx-x64.dmg., y seguir las instrucciones del asistente del instalador además de definir una contraseña en el instalador del lenguaje:



Ilustración 44. Instalador Java para Mac.

* Linux: para la instalación en este sistema operativo se debe ir a la página oficial, luego elegir el tipo de instalador dependiendo de las especificaciones, los tipos de plataformas son Linux y Linux basadas en RPM, dependiendo del caso se descarga el archivo (puede ser tipo .tar, o .rpm), una vez descargado se ejecuta y aceptan los términos y condiciones de uso.

**Anexo 4. Instalación de Spring Boot**

Spring Boot es un framework que se puede descargar de varias formas. Podemos utilizar el propio IDE de Spring Boot, el cual podemos descargar por internet, también podemos utilizar eclipse o NetBeans y con ayuda de plugin instalar Spring Boot. En nuestro caso, utilizaremos la forma más tradicional que es utilizar Spring Initialzr, una página web para generar automáticamente proyectos de Spring Boot.



Ilustración 45. Spring Initialzr

Aquí podemos definir que queremos en nuestro proyecto de Spring Boot. Nosotros utilizaremos el lenguaje Java 17 junto a Maven. La versión de Spring Boot debe ser la 3.1. El resultado final de nuestro proyecto se vera de la siguiente forma:

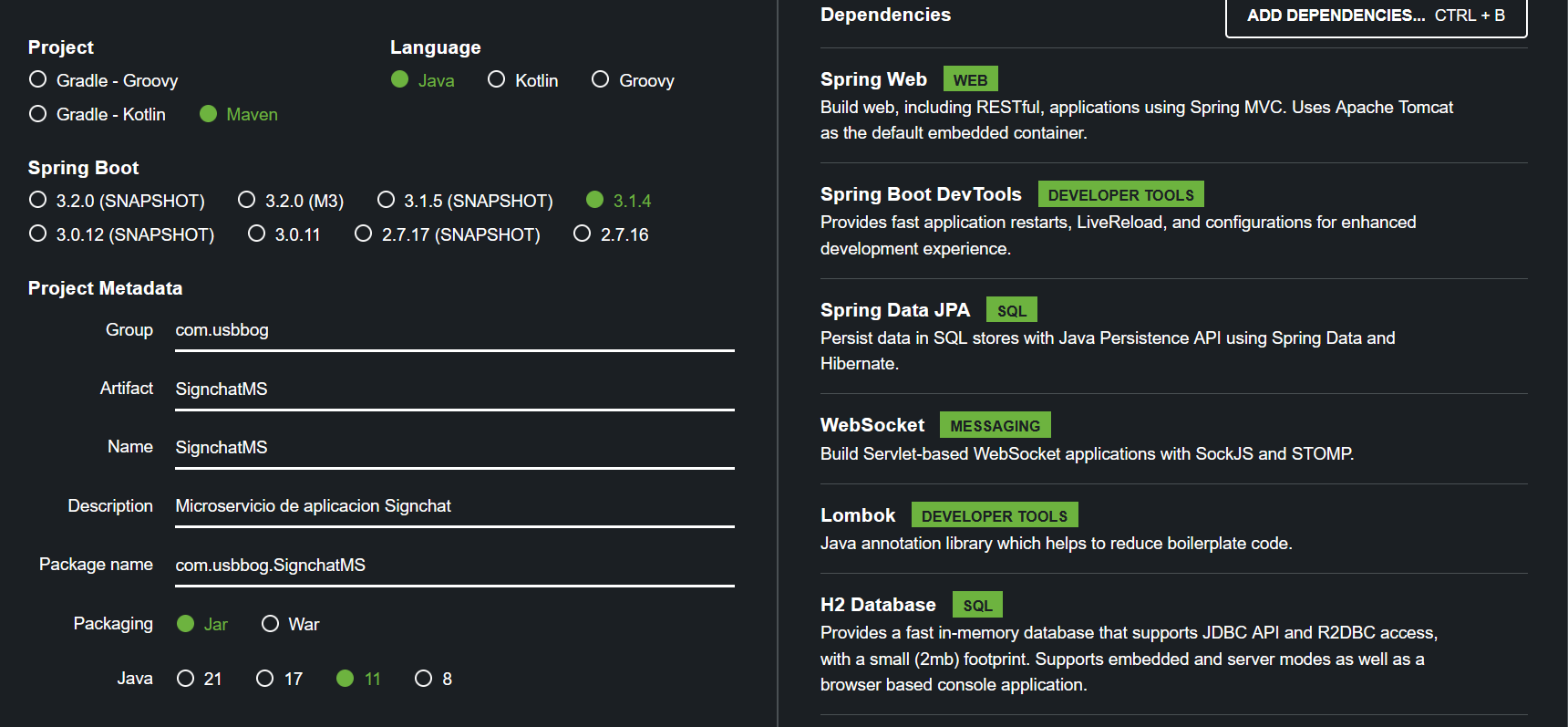


Ilustración 46. Configuración del proyecto.

Las dependencias que son mostradas en la imagen son las que utilizaremos en el proyecto. A pesar de eso, no todas pueden ser agregadas de una vez al crear el proyecto, por lo que después toca modificar un poco el archivo pom.xml para ello. Nuestro archivo pom.xml final se ve de la siguiente forma:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>3.1.0</version>

        <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <groupId>com.usbbog</groupId>

    <artifactId>SignchatMS</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>SignchatMS</name>

    <description>Microservicio del sistema de mensajeria de la aplicacion Signchat</description>

    <properties>

        <java.version>17</java.version>

    </properties>

    <dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-actuator</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jdbc</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-thymeleaf</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>

        <!-- https://mvnrepository.com/artifact/org.mariadb.jdbc/mariadb-java-client -->

        <dependency>

            <groupId>org.mariadb.jdbc</groupId>

            <artifactId>mariadb-java-client</artifactId>

            <version>3.1.4</version>

        </dependency>

        <dependency>

            <groupId>com.h2database</groupId>

            <artifactId>h2</artifactId>

            <version>2.2.220</version>

        </dependency>

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <version>1.18.20</version>

            <scope>provided</scope>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <scope>runtime</scope>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-test</artifactId>

            <scope>test</scope>

        </dependency>

        <!-- https://mvnrepository.com/artifact/org.springdoc/springdoc-openapi-starter-webmvc-ui -->

        <dependency>

            <groupId>org.springdoc</groupId>

            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

            <version>2.0.3</version>

        </dependency>

        <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-validation -->

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-validation</artifactId>

            <version>3.0.4</version>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-websocket</artifactId>

        </dependency>

    </dependencies>

    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>

        </plugins>

    </build>

</project>

A continuación, una descripción de las dependencias más importantes dentro del proyecto.

#### Anexo 5. Descarga de Swagger UI

Esta dependencia nos da una interfaz gráfica donde mostrar nuestra API REST y poder probarla.

        <dependency>

            <groupId>org.springdoc</groupId>

            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

            <version>2.0.3</version>

        </dependency>

#### Anexo 6. Implementación Spring Data JPA

Con esta dependencia, podemos construir nuestra base de datos de manera automática con código Java, sin necesidad de utilizar el lenguaje SQL.

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

#### Anexo 7. Implementación H2 Database

Esta es la base de datos embebida en la aplicación, la cual es la que usaremos principalmente para la creación de los modelos y probar la API.

        <dependency>

            <groupId>com.h2database</groupId>

            <artifactId>h2</artifactId>

            <version>2.2.220</version>

        </dependency>

#### Anexo 8. Lombok

Esta dependencia nos ayuda a ahorrar código Java, para que la aplicación quede más ligera y eficiente de utilizar.

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <version>1.18.20</version>

            <scope>provided</scope>

        </dependency>

#### Anexo 9. Implementación Spring Boot Devtools

Esta dependencia esta más enfocada en ayudar más en el desarrollo de la aplicación que en su funcionamiento. Nos otorga herramientas como recarga en tiempo real de cambios, notas de errores más concisas, advertencias de mejor forma de escribir o implementar algún código o tecnología, entre otras funciones.

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <scope>runtime</scope>

            <optional>true</optional>

        </dependency>

#### Anexo 10. Implementación Websocket

Con esta dependencia, podemos implementar Websocket a nuestro proyecto utilizando SockJS y StompJS.

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-websocket</artifactId>

        </dependency>

#### Anexo 11. Manual

Utilizando la herramienta Swagger es un entorno grafico donde se puede visualizar las funciones que existen en el API que se desarrolló mostrando la dirección que permite implementarlo, esto se puede ver en las siguientes imágenes

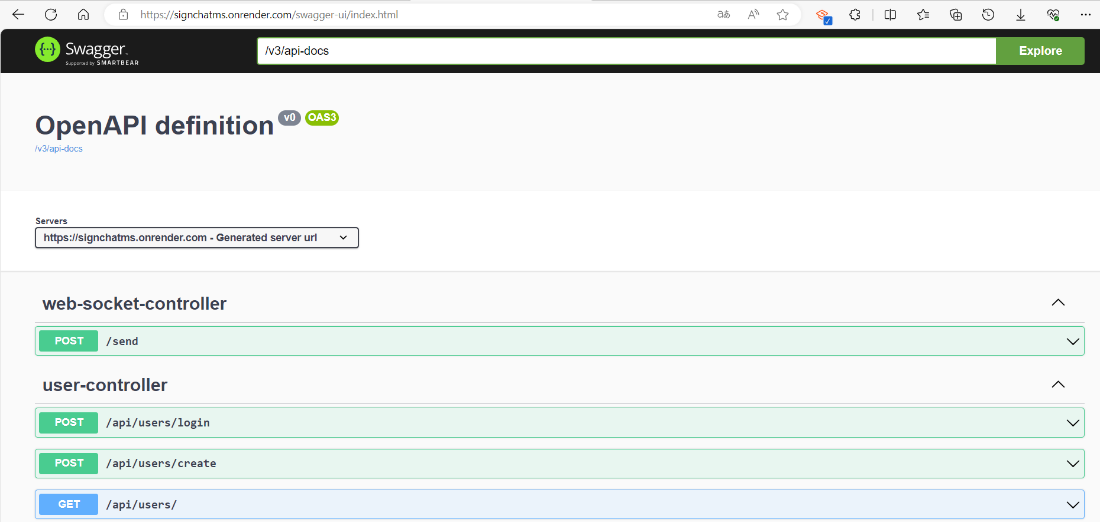


Ilustración 47. Funcionamiento Swagger

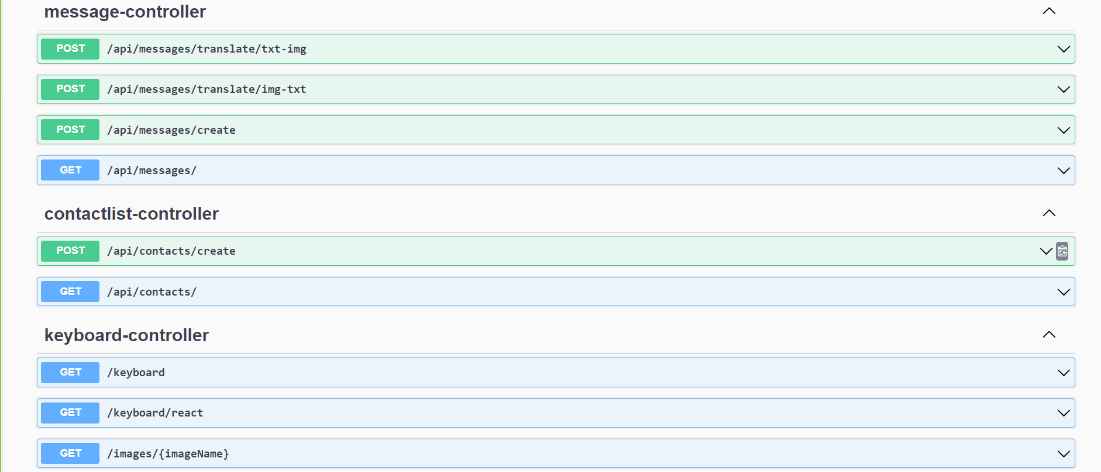


Ilustración 48. Funcionamiento Swagger

Las funciones se dividen entre las peticiones GET o POST donde GET es utilizado para solicitar información de la base de datos y Post es usado para él envió de información a la base de datos. Las actividades se dividen en las clases que se crearon según su funcionalidad.

Web-Socket-Controller:

* /send: utilizado para el envío de mensaje mediante el websocket hacia la base de datos

User-Controller: funcionalidades creadas para realizar pruebas de funcionamiento y pueden ser modificadas o alteradas en base a las necesidades. Además, que las funciones se destinan a la creación de usuarios

* /api/users/login: envió de mensaje mediante el uso del websocket del api desarrollado
* /api/users/create: en una actividad de prueba para la creación de usuarios y almacenamiento en la base de datos de SpringBoot

Message-Controller:

* /api/messages/traslate/txt-img: Esta es la base principal del api pues esta función se destina a cumplir la función de transformar los textos ingresados por su visión en lenguaje de señas
* /api/messages/traslate/img-txt: Esta funcionen es el complemento de la función principal que se encarga de la traducción de los textos expresados en imágenes para convertirlos a texto
* /api/messages/create: función destinada a crear y guardar el texto desarrollado
* /api/messages/: esta función es de tipo GET con la finalidad de obtener los mensajes que se han almacenados

ContactList-Controller: funcionalidades creadas para realizar pruebas de funcionamiento y pueden ser modificadas o alteradas en base a las necesidades. Además, esta se encarga de listar la lista de contactos según el usuario.

* /api/contacts/create: se encarga de crear una lista de contactos
* /api/contacts: lista los contactos

Keyboard-Controller:

* /keyboard: esta funcionalidad es utilizada para la solicitud e implementación del teclado virtual el cual publica imágenes individuales en los botones según la letra especificada esto mediante el uso de la funcionalidad /images/{imageName}.
* /keyboard/react: realiza la misma actividad de /keyboard pero esta funcionalidad destinada para ser utilizada en frameworks para desarrollo web avanzados
* /images/{imageName}: esta función es la encargada de buscar las imágenes individualmente que están almacenadas además de ser utilizad por la función /keyboard

También se puede visualizar la estructura de la organización de la base de datos desarrollada para pruebas en el apartado de Schemas:



Ilustración 49.Swagger estructura de Base de Datos